目錄

前言

2015年4月，我在一个国内交付项目上，比较“全程”的跟踪了一个项目从启动到进入开发阶段的整个过程。我们根据售前阶段的输出对项目有了大致的了解，然后从北京飞到深圳，和客户一起进行了为期一周的需求梳理和项目计划，与此同时，用户体验设计师负责梳理信息架构，原型设计，最后形成视觉设计稿（唐婉莹负责视觉设计）。

与以往的经验不同的是，整个过程我们和客户（需求的提出方）一起合作，指定计划，并快速归纳、指定出一个可执行的方案。在需求梳理阶段结束的时候，我们形成了一个项目计划，重要的是，我们知道了这个项目最终会服务于谁，谁会从中受益，项目进行过程中有哪些风险，有哪些资源限制，为什么要做这样的决策而不是相反等等。

不得不说，这个过程在一定程度上颠覆了我对软件开发的认识。如果说软件的生命周期是一本书（分为上中下三册）的话，平时我所理解的开发则跳过了书的前几章，以及“中下”两册。我们涉及的一些工程实践，开发方法论，都是围绕着中间少的可怜的一小部分。我们根据看到的内容，会对缺失掉的前几章进行浅薄的猜测，而对于更多的看不见的后半部，则充满了更多的未知。

软件开发

毫无疑问，软件开发是一系列庞杂，复杂的活动的集合。人们现实世界中的发现问题，并尝试通过软件的方式来解决这些问题。

一个广为流传的段子是：“我有一个绝妙的创意和一个靠谱的团队，就差一个写代码的了”。抱有这种想法，想要在互联网时代获得商业成功的，也只能祝他成功了。

产品当然不可能简单到只要创意和开发就可以完成，甚至创意本身，在产生之初，也只是一个粗略而缺乏考虑众多细节的“种子”，它需要在专业的设计师的引导下，完成一系列的发散，收敛，探索，验证而形成一个可行的方案（未必最优，但是需要切实可行）。最后软件开发人员再来开发原型，再投放给真实用户做测试，然后回过头来再影响设计决策，周而复始。

一个项目，从开始有粗略的想法，到可以开始编码交付，需要经过十分艰辛的过程。对于开发者来说，这个过程事实上是很难看到的：需求来自于一个神奇而牛逼的团队（当然，有时候当需求比较费解的时候，开发会在心里骂人）。而对于参加项目初期的需求梳理，引导过程的设计师团队来说，设计结束之后，他们有需要转战下一个项目，项目如何落地的细节则未必清楚：反正有一个神奇而牛逼的团队来负责就是了。

在我们看来，一个产品既是迭代产生，逐步成熟的，也是有章可循，有很多工具和方法来支撑的。如果只是粗略的划分一下，可以分为两个方面：

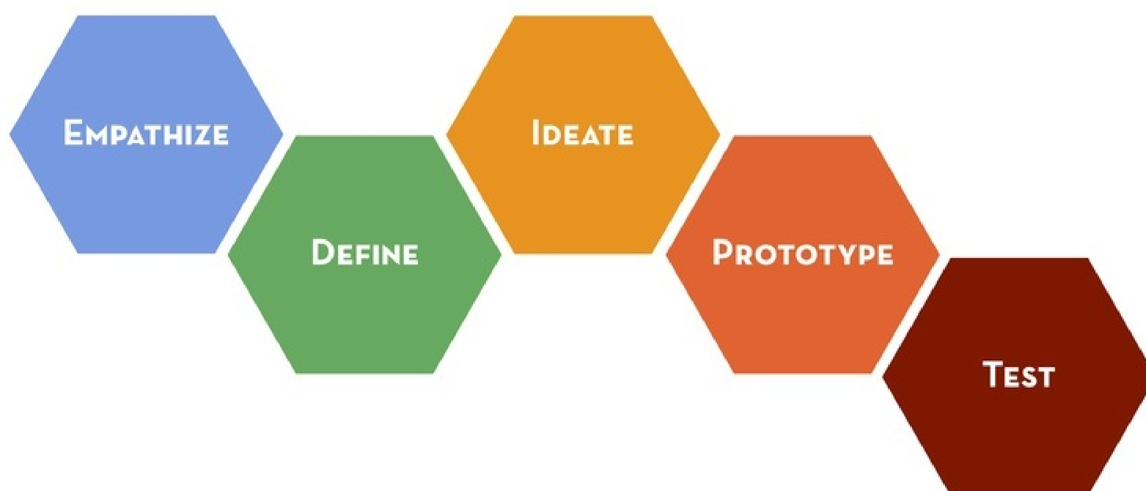
- 如何将产品原始的想法具体化

- 如何快速、忠实地实现具体这些需求

通过设计MVP来验证一个idea的可行性、通过不断地测试、从测试结果中学习的方式的已经为很多企业，团队，创业公司采用。但是在具体操作上，如何将一个idea恰如其分的变成可供交付的MVP，如何调整方向/策略进行下一个迭代，如何将软件开发中的工程实践应用在整个过城中，目前还没有普遍适用的“规则”。

使idea具体化

斯坦福大学的D.School的设计思维（Design Thinking）在产品设计甚至在软件开发上都产生了深远的影响。设计思维脱胎于传统的设计方法，不过更加强调设身处地地为最终用户考虑。



除了传统的设计方法：

- 发现问题
- 发散收敛（头脑风暴尝试众多方案，根据现有资源收敛出方案）
- 原型
- 测试验证

设计思维加入了移情（Empathize）：

- 移情
- 定义
- 发散收敛
- 原型
- 测试验证

看起来只是多了一个步骤，但是正是这一点体现了以用户为中心的思维方式。

实现idea

具体到实现一个想法（或者说一个原型）时，你会遇到各种各样的选择。选择什么样的技术栈，服务器部署在何处，数据如何存储，产品的安全性如何考虑，对性能的要求是什么样的等等。

不幸的是，做出这些决策仅仅是万里长征第一步。你还会遇到很多其他问题：软件质量如何保证，当进度变慢时如何应对，如何划分成员的角色，如何确保各个模块集成时发生重大问题等等。

相信我，鼓吹“只差一个程序员”的创意是落不了地的。围绕着这个可能会改变世界的创意，你需要一系列的实践，技能，工具和方法。

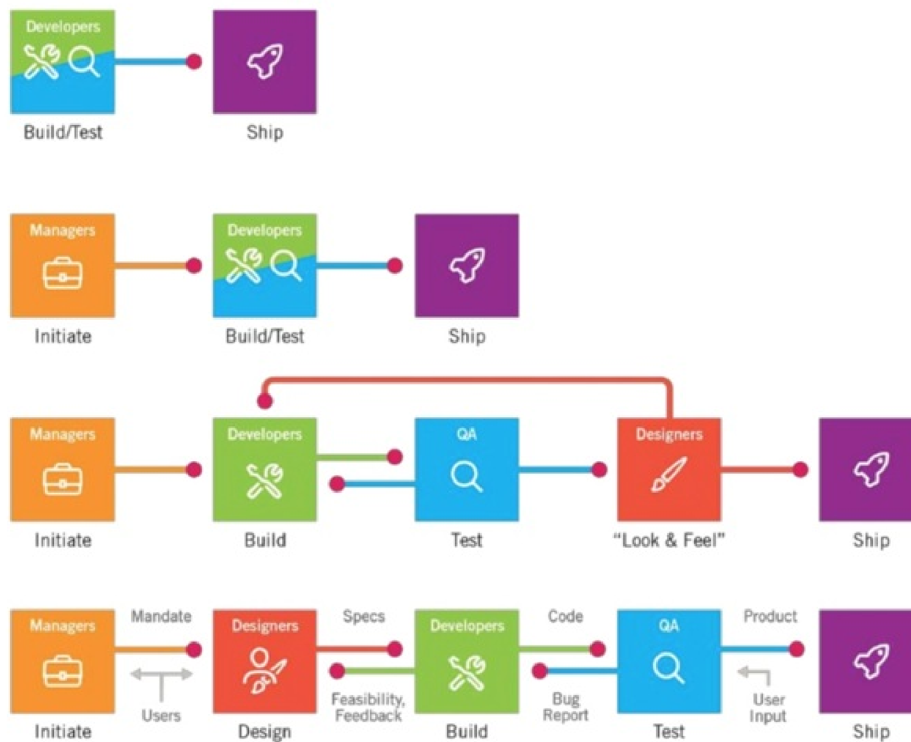
- 基础设施
- 自动化测试
- 自动化配置
- 可视化的开发流程
- 知识分享
- 高效的开发流程

这些看似相关度不是很高的条目，会帮助你在落实一个创意时发挥作用，并让你的创意变为实际的产品。

传统上，人们认为软件开发可以割裂成不同的阶段，就像建筑工程一样。设计，开发，测试，上线，运维。然而事实证明，软件开发中的变化比建筑工程要频繁得多，而且变化的程度也更具颠覆性。这需要通过更小的迭代节奏，和更紧密的合作才有望改善。

正如《About Face: The Essentials of Interaction Design》一书中这张图所表达的那样， 设计师，开发，测试工程师，运维工程师需要一起协作，小步前进。在验证过程中不断学习，以期最终完成产品的定义和实现：

Software development has evolved



Page 8

Courtesy of cooper CC 4.0 BY-ND

From *About Face: The Essentials of Interaction Design* (Fourth Edition. Wiley, 2014)

用户研究

用户研究之前应该知道的事

追根溯源

如果我们要说用户研究，就先得追根溯源，谈谈它的上游以及整个软件时代的大背景。

80后90后70后的爸妈们年轻时大多数是农民和工人，而到了我们这一代，所见的职业很多是白领蓝领，这正好说明了一个历史状况：从70年代开始，我们正从大机器生产的工业时代进入到以信息和知识为主的后工业时代，人们的职业从以前的手工业者转向政府部门、教育机构等白领职业，软件就在这样的时代背景下诞生的，它是科学进步的产物。

软件既然是科学技术的产物，必然不会是门槛很低的东西，一开始它是给科学家、军工部门和政府部门用的，用户是有技术背景的专业人士，目的是为了提高工作效率和进行严格的控制和管理。随着时间的发展，一些大的企业也需要用软件来提高生产和管理效率，软件此时从一个专业化的工具变成了代替人来工作的工具，随后又变成了能够给企业提供业务数据参考以帮助其发展的工具。与之对应的用户也从以前的技术人士变成了普通的工作人员。

技术人员使用的软件如果给到普通人去用，肯定会非常困难。这时候交互设计就出现了，它使得软件对于普通人来说，就像是一个神奇的魔术盒子，只要你按下按钮，就可以得到你想要的东西。

即使这个东西背后需要花一万个步骤，你也不必知道，完全以结果和目标为导向。所以交互设计是以目标为导向的行为设计，首要的步骤是要了解用户使用产品的目标，接下来需要在人们的行为习惯和软件的操作流程之间搭一座桥，使得用户可以顺畅低达到他们的目标。

在软件发展的初期，人和软件的关系是以软件为主的，因为软件开发的难度和时间成本很高，没有办法根据领域的不同来设计不同的软件，一个企业内部管理软件，它再难用也得用下去，没得选。公司对于效率和员工的心情上，必然会趋向前者。到了后来，技术的不断发展使得软件开发难度降低，速度加快了十几倍，使得企业对于软件有了很多的选择，并且软件扮演的角色从幕后走向了台前，客户第一接触的不是人，而是软件，所以软件的易用性成为一种竞争优势，同时内部软件的易用性也更近一步提高了员工的工作效率。所以此时软件和人的关系变成了以人为核心，易用性变成了企业开发软件口头禅。既然服务对象是人，当然用户研究、视觉设计也随之出现了。

用户研究不是救命稻草，了解它的作用比使用它更重要

用户研究并不是一个好名字，它让人觉得它的研究对象就是用户，即“使用产品的人”。但其实它包含的研究非常多样化，包括利益相关者、顾客、用户 、专家、竞品调研和市场、策略、业务领域调研，并且这几种调研背后对应的目的是不一样的。比如，利益相关者和顾客的调

研，常常是为了更好地寻找商业模式、节省资源；用户的调研常常是为了培养用户同理心、提供设计参考；做市场、策略调研是为了确定产品的范围和用户的水平、技术的实现的程度等。

但并不是调查得越多越全越好，而是要清楚地知道自己需要什么，不需要什么，来选取调查内容。 调研所提供的价值范围限于以下几点：

- 作为一个理解的工具，帮助设计师培养同理心
- 作为管理工具，统一团队的意见，为设计提供参考依据
- 帮助设计师了解产品相关的业务和市场的上下文
- 能给不同的人提供不同的商业灵感

所以这就回答了一个问题：为什么大的互联网公司有用户研究，但是却不一定能做好产品，小的创业公司没有用户研究，但也能做出优秀的产品。因为用户研究是一种工具，有它不能救活一个产品，没它不会生产不出产品。比如一个创业公司所做的产品是针对90后，而团队本身也是90后，他们就不需要培做用研来养同理心；团队人少，意见容易达到统一，就不需要用户研究来当管理工具；ceo本身对市场和物业非常熟悉，就不需要了解更多业务，而应该把重心放在传达业务知识给团队成员上。

使用户调研失效的魔法

正确的认识加上正确的使用方法才能让一个工具真的发挥作用，用研作为一种工具，有它明显的“使用规范”。

从它的功效来看，第一，“作为一个理解的工具，帮助设计师培养同理心“，也就是说，设计师需要做一个90后使用的产品，最极端情况是把自己的思维行为变成90后，这样才能准确无误把握他们的需求。这就明确地规定了玩法：设计师不可能让用户研究人员去传达研究结果，因为这样的传达是没有办法培养同理心的，设计师还是会按照自己的想法设计一个产品。

第二，“能给不同的人不同的商业灵感”，意思是说在调研的过程也是一个极好的激发灵感的过程，创意不应该是一个流程中的一环，应该贯穿到整个流程中去。如果是由用研人员传递研究的信息，我们就缺失了一个大好的产生灵感的途径。并且，用研人员不是设计师，他不知道设计师需要哪些原材料，很可能会遗漏，或者完全没有抓住研究方向。而方向很可能不是一开始所计划的，而是随着调研的深入不断具象化出来的。

解决这两个问题的方法很简单：要么是让设计师自己去做调研，要么组建一支跨学科团队，让设计师和用户研究人员一起调研。

大的互联网公司的问题就在于，他们虽然有用户研究，但是由专人去做，没办法培养同理心； 至上而下的团队结构没办法成为统一的管理工具； 设计师能了解到的上下文非常有限，也不会作为工作内容之一。 设计师通常不参与商业策略研究，不用研究市场。

用户研究的内容

用户研究的内容，可以分为三层：

第一层是总体的研究，比如中国人消费习惯，上班族使用移动设备的时长等，这些研究我们不用去做，小团队研究出来的结果也不可靠，它需要非常专业的统计学知识，所以这类的研究可以通过专业渠道买资料或者是寻找公共资源来阅读即可。

第二层是目标性的研究，我们需要搞清楚，在这个行为之后，人们的诉求是什么。人们的行为时常发生改变，但人的目标随着时间可能不变化，或者变化很慢，例如从古至今我们对于沟通交流这个需求从未改变，只不过一直在用不同的方式在满足：飞鸽传书、电话、微信等。我们应该把用户的行为抽象化来思考。

第三层是环境和行为的研究，这个研究的内容是我们真正需要着力的点，它会根据环境、文化、心理、状态、时间的不同而发生变化。

用户调研对设计师的价值

其实，用户调研对于设计师来说只是一个工具，设计师的价值大部分不会体现在这里。例如，明星产品一定不是“设计师”设计出来的。这里的设计师指的是现在在市面上的视觉设计师、交互设计师、用户体验设计师。

我所理解的好产品一定会拥有超强的基因，在短时间内，被广泛接受并且自传播。这种产品一定不是设计出来的，而是长期孕育出来的，由长期在这个行业的佼佼者所创造，融合了他们过去的经验和未来的展望。uber的CEO卡拉尼克，经历了8年的创业失败，才创造出了一款让全世界为之一惊的移动约车产品，他成功是因为创始人了解所有的商业模式，有资源能力、有坚定的意志力，当下的市场也刚好成熟，所有条件都指向一个可以成功的方向，这个事情才有可能做成。

所以，每个行业的佼佼者才是最好的设计师，世界上的好产品都是由他们创造的，而只会做用户调研、画界面原型的设计师，从来不是产品的主人，而是产品的服务者。

我记得有一次给一群游泳场馆经营者做游泳相关的app产品咨询，提供产品策略以及IT实施咨询，当我们激情慷慨说完方案以后，其中一个客户说：“你们说的这些我都知道，产品我自己也有了，我不知道为什么还要照你们的思路再做一个。”当时我们找了很多理由，最终谁也没有说服誰，但是我知道也许他是对的。这位老板每天花2-3个小时时间观察他的游泳馆，想过无数办法解决冬季场地空置的问题，走访调查过国内外的游泳馆，相信没有人比他更了解他需要的是什么，比他更清楚他所面临的问题及原因是什么。

而作为咨询师的我们所做的只是花了两周的时间进行用户访谈和调研、参考了国内外的资料、找到了游泳相关的基础调研数据，居然就能拍着胸脯说我们懂游泳行业，其实我们不懂，只是学习的能力快，比一般人要懂一些而已。

说这些并不是要表达设计师没什么用，而是要说，设计师要看清自己的定位，要了解到用户调研的局限性，我们没有办法站在客户肩膀上说话，我们只是助推器一般的角色，用我们的方法和经验，加速客户研发产品的速度，保证最终产品与初始想法的一致性。

未来模式

未来，软件开发成本一定会逐年降低、开发周期一定会逐年缩短，这就意味着，有可能各个行业的佼佼者也可以很轻而易举地创造出有价值的产品，而不再依赖设计师。举个例子，一个优秀的高中老师想要根据往年的教学情况和现在教学情况的对比，预测学生们考上各大院校的几率，但是学校和市面上都没有这么一款软件，于是他想自己开发一个:把往届和本届学生每次模拟考试的结果从学校数据库里面抓取出来，从各个纬度（及格率、优秀率、分数线趋势等等）做对比，就可能预测结果。理科背景的他从网上找了几个免费的数据软件组装起来，再用一个简单的合成软件把它们连接在一起，按照需要改动几个个性化的小设置，这个软件就可以跑起来了。他试着把数据导进去，眞的能够显示他想要的几个预测结果，经过几个版本的迭代，这款软件已经90%能满足需求。

这个情况看起来很遥远，但是在10年以后，软件的开发成本很低、市面上有很多可组装的软件、实施周期很短，那些伴随着电子产品长大的孩子也走出社会，他们完全有能力做这样的事情的。到了那个时候，已经不可能存在用户调研的工作、也没有视觉设计师和交互设计师，也许会有新的职业叫做软件组装设计师，来为企业组装更为精细复杂的软件。

所以，如果现在的设计师想要在更远的未来保持高傲的姿态，就必须把设计师的身份模糊掉，要是喜欢某个行业，就深度了解这个行业的方方面-商业模式、技术、用户，积累经验和资源，做行业佼佼者；要是喜欢跨行业做咨询，就要要求自己有很好的理解力，最大程度地理解他们的业务和需求，不断积累案例，提供给客户更宽的视角和丰富的案例。

竞品分析-产品

竞品分析是在做产品的过程中频率最高的分析之一，它常常用来帮助我们获得灵感、了解我们自身产品的不足、挖掘新的机会等。如果我们能用合理的方式来分析竞品，将会为产品提供一盏指路明灯；相反的，如果我们只是简单粗暴地对比分析，拿到结果直接指导设计，则会把产品带上歧途。

我常见的竞品分析有那么几种：

1. 直接复制粘贴：找到【各种相似】的竞品，把它们和自家产品对应的功能摘出来，照抄；
2. 取共性复制粘贴：拿市面上【标杆性】竞品做比较，找出他们共有的功能，认为别人家都有的自己必须有，抄之。
3. 融合+创新：把市面上【大部分】竞品拿过来做比较，把它们的可取之处找出来并揉在一起，在此基础上做一些适配自己产品的“创新”，让用户觉得既熟悉又有点新鲜，俗称“微创新”。

这些分析方法的初衷都是想直接从竞品分析中找到一个确定的答案。原因是自己研发创新是非常耗费时间、人力、财力的，如果能有一个市面上已过验证的功能或产品直接拿过来使用，剩下的事就只有市场运营了，将节省大量的资源，迎着风口快速成长。如当通讯软件whatsapp在2011年一炮而红过后，类似产品如雨后春笋：微信、line、viber、kik等等；魔法相机一夜走红后，face U、脸萌也应运而生。其实这种方式并不是不好的，相反的，这是一种相对聪明的商业手段。试着想想看，如果不是腾讯简单粗暴地抄袭了whatsapp，我们不可能那么快有微信那么方便的沟通工具，也不可能有微信今天如此强大的功能。我们就像偷了别人的武功秘籍，让全民都学会了轻功一样，做了一件利人利己的事情。到今天为止，一些大型互联网公司还在利用这一套路，被称为“山寨公司”也在所不辞，因为在决策者的眼里，不断抄袭别的产品，再将自己的用户流量注入到产品中，是对公司成本的极大节约。

互联网发展到今天，市场不断走向成熟，这种方式的成本也逐渐提高了。原来抄3个产品，就有1个能够赚到用户流量，而今天做10个产品都不一定有一个能活下来。当下，用户面前已经摆放了太多选项，如何拨开眼前横七竖八的相似产品，让自己发出不一样的气味吸引到用户，成为每个【后起之秀】必须要努力的事。

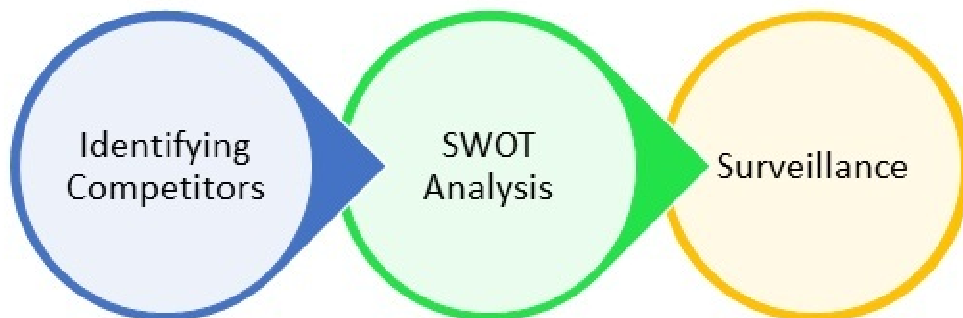
我们先来看下正确的竞品分析将为产品带来哪些好处：

- 竞品分析能和用户研究互做补充。用户研究都是关于“人”的研究，必定带有大量主观性，而竞品分析可以从“别的产品”的视角来反思自家产品，让这种主观性得到很好地控制。
- 确定市场机会。当某个产品在市场上赢得用户的时候，一定是打开了一片新的市场，无论市场是大是小，而竞品分析能让我们知道这块蛋糕有多大，还剩多少可以分到自己碗里。

- 利用竞品分析分辨目标用户。目标用户的确定对一个产品来说至关重要，它是我们做产品需要的第一颗定心丸，以后的所有功能都以此为基础。我们能从竞争对手那里知道我们的用户大概是那些人，不是哪些人，可以在此基础上挖掘属于自己的用户分类。
- 发现新的竞争对手。竞品分析让我们不断挖掘新的竞争对手，甚至一些不直接竞争但是在某方面做得不错的产品。例如要做共享经济的旅游住宿产品，大家都会想到airbnb，但进一步挖掘国内市场，会发现像【小猪短租】中国版airbnb也做得不错，再进一步挖掘，发现自如租房这种针对城市上班族租房市场的产品，里面嵌套的【管家】服务，也可以拿来借鉴。

如何做竞品分析

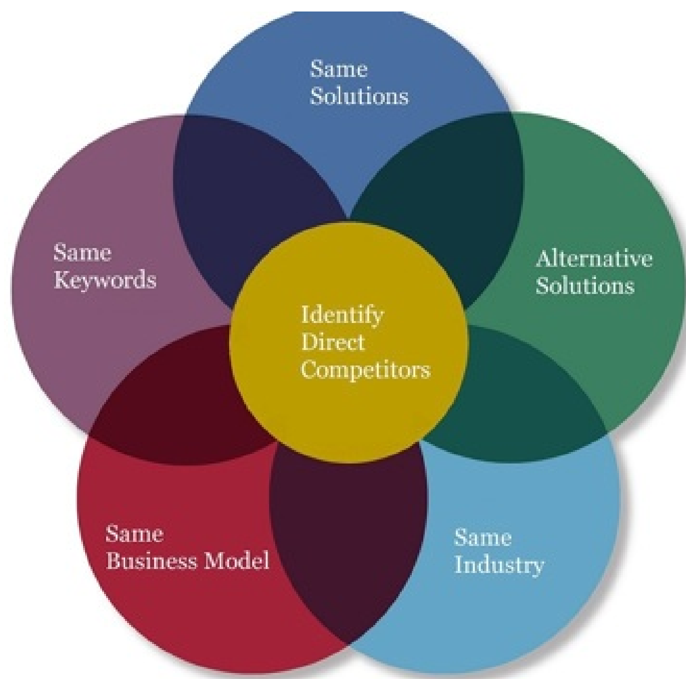
识别竞争对手--分析竞品--持续分析



1.识别竞争对手：你是谁，谁是你的竞争对手

如果你是在一个成熟市场中的寻求发展的产品，识别谁是你的竞争对手就犹如拜老师或者交朋友一般谨慎，因为它们将会影响你接下来走的每一步。

我们首先要做的是正确的评估自己与竞品中的区别。一个产品成功一定是多方面因素影响，切不可光看中竞品分析中的某一点，而是要做全面的对比分析。所以我们可以挑选一些差异大的或者有很深影响的纬度进行对比：资源、用户量、产品定位、使用场景、目标...等等。如下图：



下图是一个想要做银行卡【优惠】的产品，产品决策者把市面上最火的微信、支付宝、大众点评、美团、掌上生活、卡惠作为竞争对手。这几个产品有非常类似的地方，所以我们把相似的竞品先进行了合并分析，然后再按照场景、用户行为、内容体量、卖点进行了对比，用浅红色代表相似度高，深红色代表相似度较高，蓝色代表相似度低。通过对比可以清楚地发现像微信、支付宝这样的产品几乎没有太多参考性；而像掌上生活、卡惠这样的产品虽然不是市场最热的，但却是最直接的竞争对手，参考价值最大。

	场景	用户行为	内容体量	卖点
支付宝/微信支付	支付/社交	被动发现 主动寻找	商户/用户 基数大	关联 商户
大众点评/美团	商户资讯 / 优惠	主动 寻找	商户/用户 基数大	关联 商户
掌上生活/卡惠	卡资讯 / 优惠	主动 寻找	商户/用户 基数较大	关联 银行卡
某产品	卡资讯 / 优惠	主动 寻找	商户/用户 基数小	关联 银行卡

这样的分析能帮助我们冷却发热的头脑，认清自己是谁，谁是你的直接/间接竞争对手。

2.分析竞品：介绍几种常用的分析工具

SWOT模型（Strengths、Weaknesses、Opportunities、Threats）

是一种分析产品/公司内外部竞争环境的工具，S、W指产品内部有哪些优劣势，O、T指的是产品外部市场存在哪些机会和威胁。此工具的优点在于它可以对所有类型的产品进行粗略的分析，既可以是实体的商店，又可以是线上的互联网产品。缺点在于力度比较粗，对于产品特质的抓取不是特别有力。



一个位于市中心的新品牌披萨店的例子：

Strengths（内部优势）：

- 产品：拥有优质的食材&配方
- 价钱：比其他流行品牌便宜
- 设计：精美的、有设计感的托盘
- 氛围：店内环境舒适，适合交谈

Weaknesses（内部劣势）：

- 品牌：不被大众熟知
- 员工：未经过专业的培训
- 经验：缺乏经营的经验

Opportunities（外部机会）：

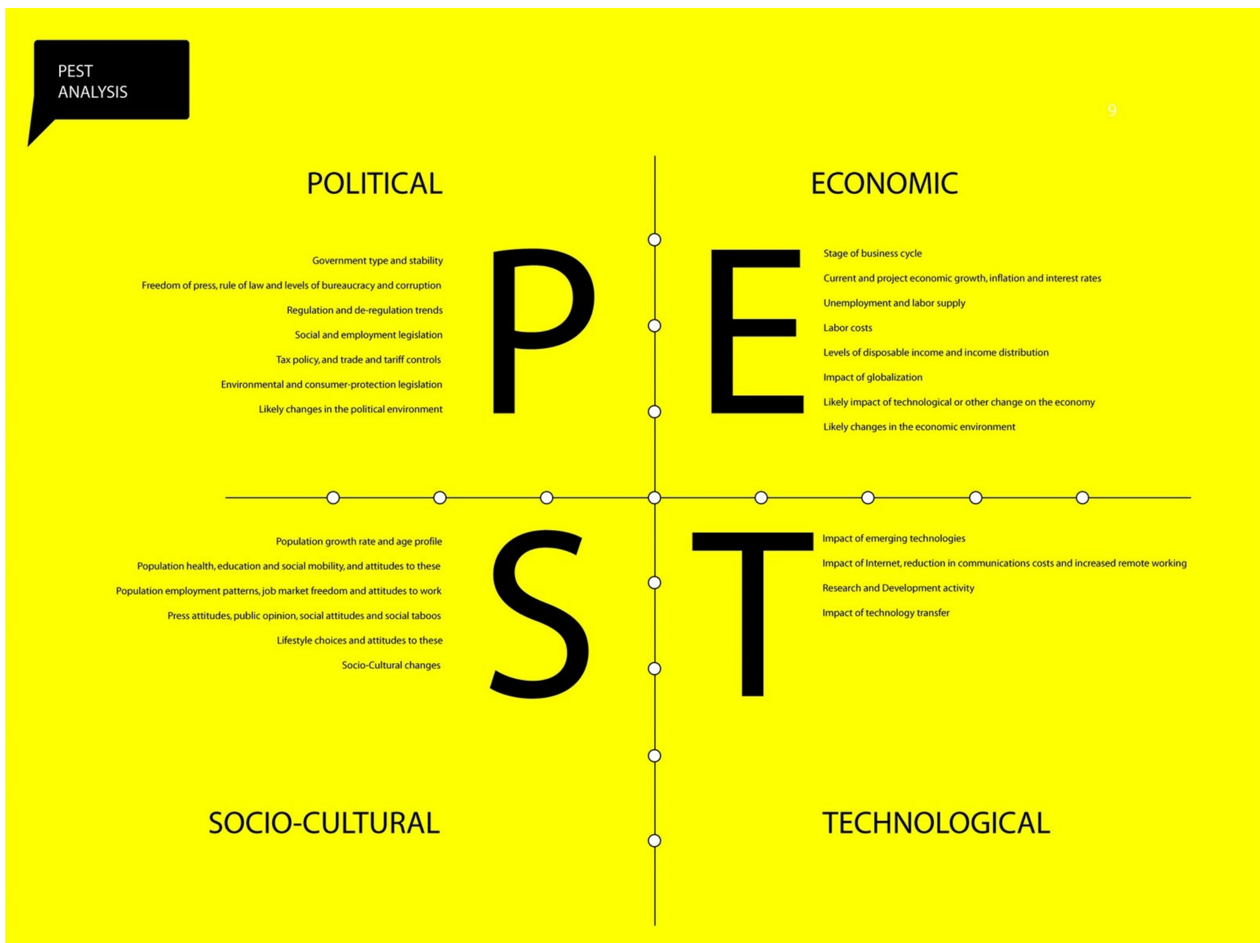
- 受众：价格和设计吸引了许多年轻的女性
- 周边品牌：有杂志、服饰、图书等资源，可发展成为综合性的门店

Threats（外部威胁）：

- 对手：竞争对手能提供更多品种的披萨
- 环境：周围500米有同类型的披萨店

PEST模型（Political、Economic、Social、Technological）

是一种产品的宏观环境分析工具，即政治、经济、社会、科技，这些因素不受产品的的改变而改变，是对产品的孕育环境的了解，和公司的运作管理更相关。



下面有些具体的例子进一步说明：

Political（政治因素）：

- 贸易政策
- 股东需求
- 国家政策

Economic（经济因素）：

- 国外经济趋势
- 本地经济趋势
- 外汇比率
- 关税

Social（社会因素）：

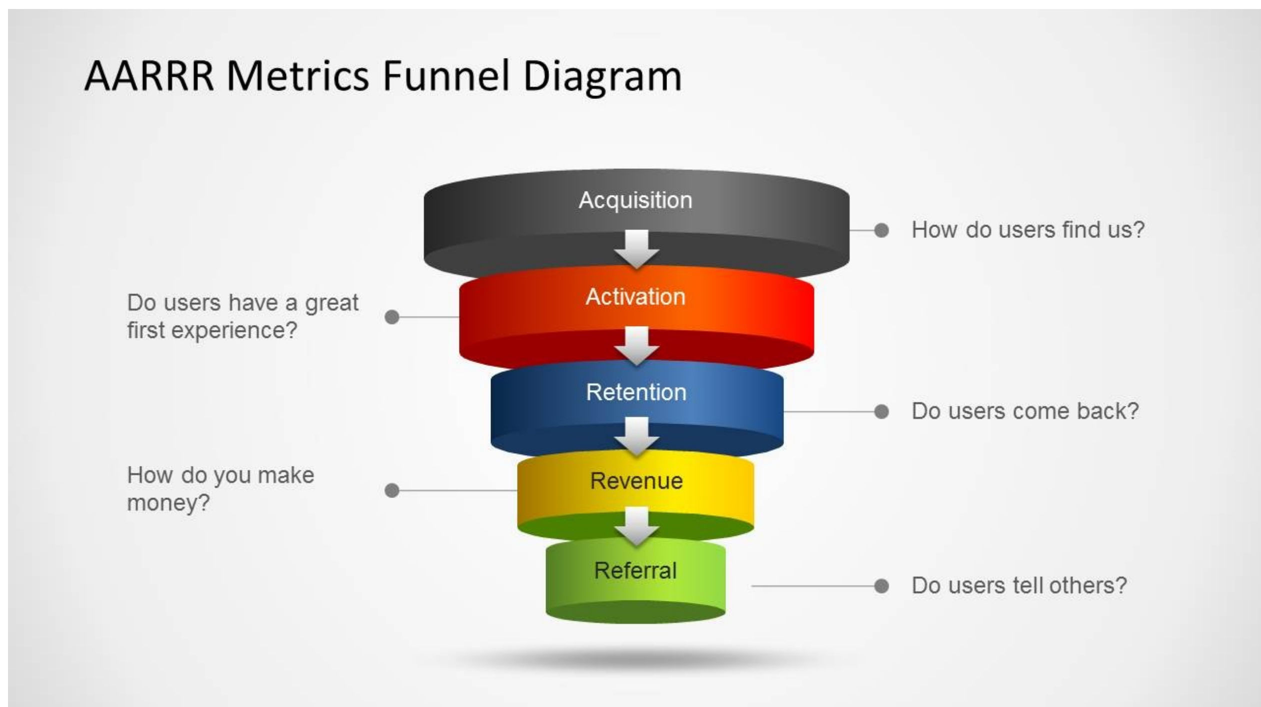
- 宗教因素
- 地域消费行为模型
- 地域人口统计
- 地域教育程度

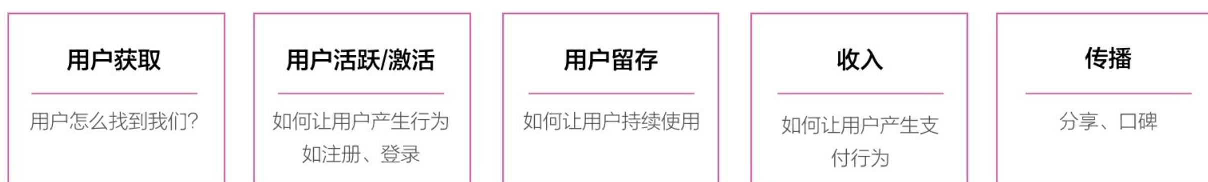
Technological（技术因素）：

- 技术发展趋势
- 技术授权/牌照
- 技术的成熟度

AARRR模型（**Acquisition**、**Activation**、**Retention**、**Revenue**、**Refer**）

是一种分析产品运营情况的模型，分别是：获取用户、提高用户活跃度、提高留存率、获取收入、自传播五个方面。比较适用于分析偏运营的产品。





AARRR模型它关注的指标，可做参考：

- 获取用户：新增用户数、CPA
- 提高用户活跃度：AU（活跃用户）、活跃率、使用时长、启动次数
- 提高留存率：次日留存率、周、月留存率
- 获取收入：ARPU（平均每位用户收入）、消费用户比例、LTV（生命周期价值）
- 自传播：K因子

Lean Canvas

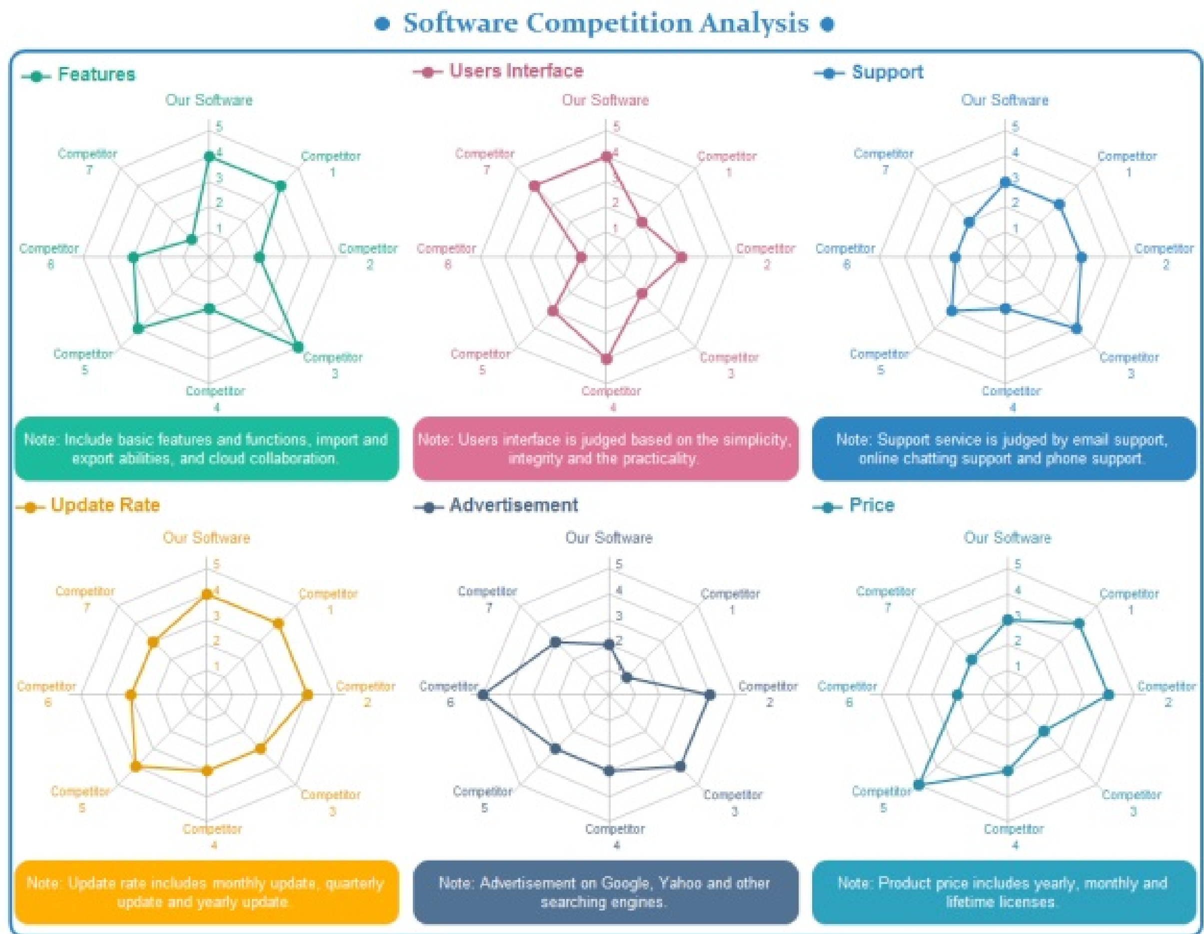
是一种全面分析产品的方式，它包括了产品应该关注的方方面面。

<table>
<tr><th>问题</th><th>解决方案</th><th>独特卖点</th><th>竞争门槛</th><th>客群细分</th></tr>
<tr><td rowspan="3">最需要解决的
三个问题</td><td>产品最重要的
三个功能</td><td rowspan="3">用一句简明扼要但引人瞩目的话阐述为什么你的产品与众不同，值得购买</td><td>无法被对手轻易复制或买去的竞争优势</td><td rowspan="3">目标客户</td></tr>
<tr><th>关键指标</th><th>获客渠道</th></tr>
<tr><td>因该考核哪些
东西</td><td>如何找到客户</td></tr>
</table>

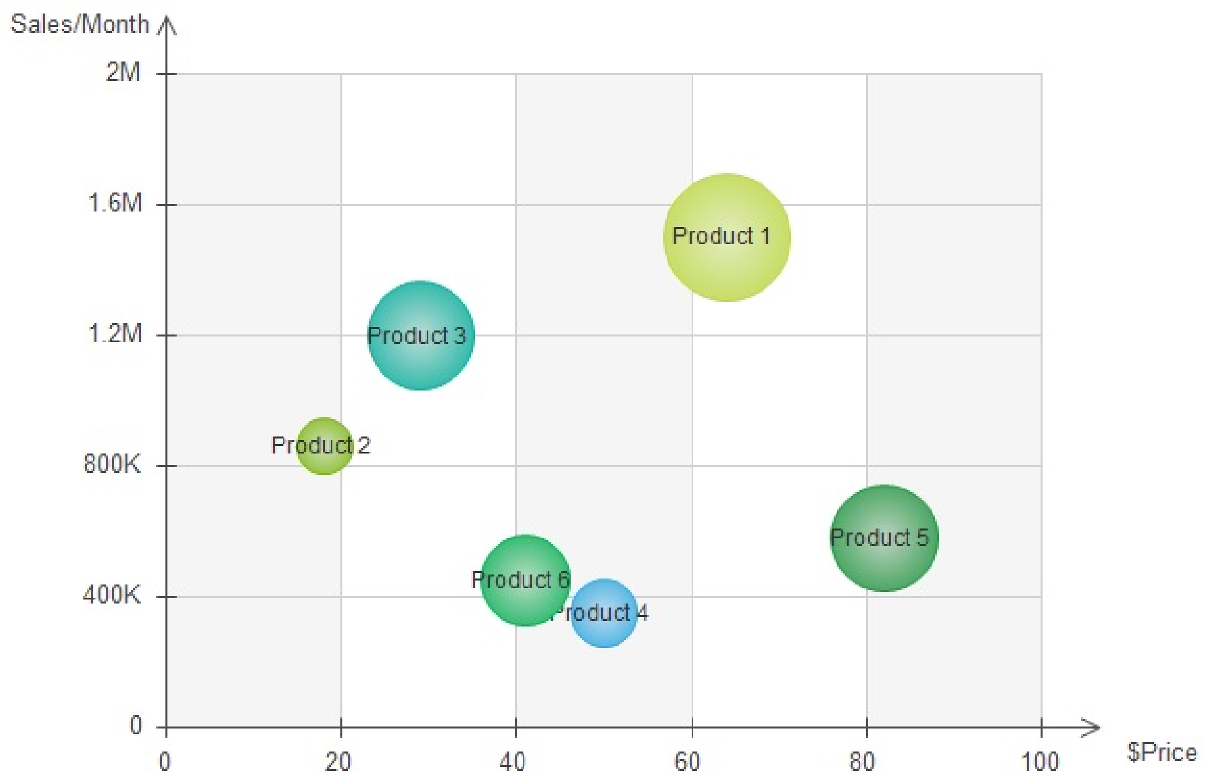
<table>
<tr><th>成本结构</th><th>收入现金流</th></tr>
<tr><td>争取客户所需花费、销售产品所需花费、网站架设费用、人力资源费用等</td><td>盈利模式、客户终身价值
收入、毛利</td></tr>
</table>

你的产品也许不适用上面的模型，那么你可以自己制作一些图表比较：

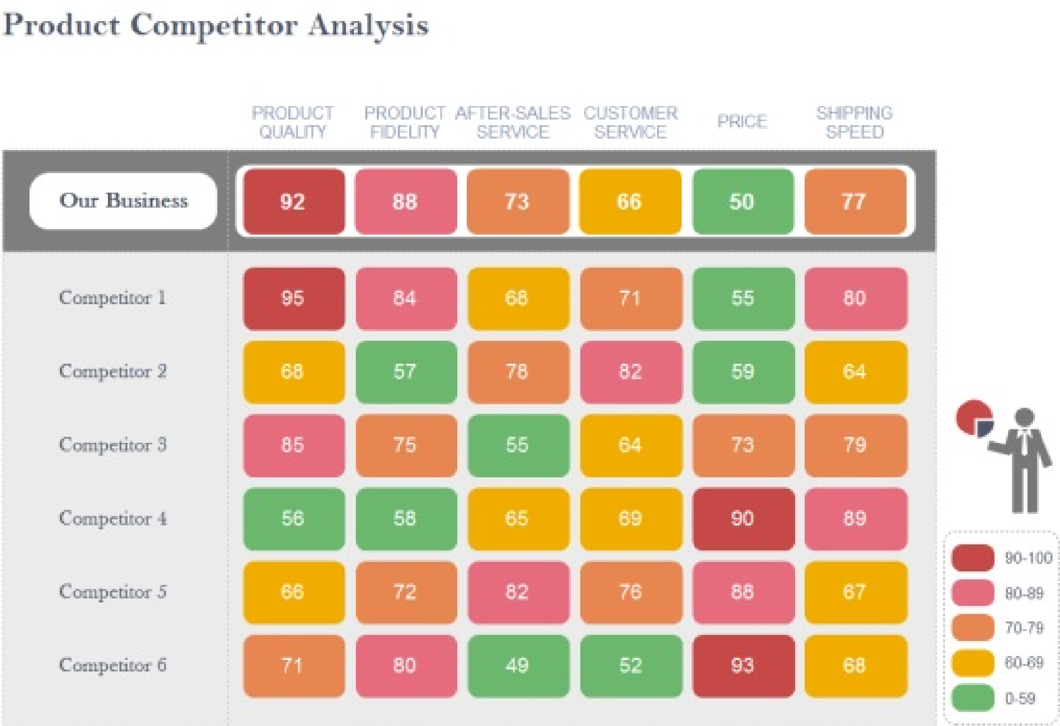
雷达图：


Software Competition Analysis
Features
Users Interface
Support
Update Rate
Advertisement
Price
Our Software
Note: Include basic features and functions, import and export abilities, and cloud collaboration.
Note: Users interface is judged based on the simplicity, integrity and the practicality.
Note: Support service is judged by email support, online chatting support and phone support.
Note: Update rate includes monthly update, quarterly update and yearly update.
Note: Advertisement on Google, Yahoo and other searching engines.
Note: Product price includes yearly, monthly and lifetime licenses.

气泡图：


Sales/Month
2M
1.6M
1.2M
800K
400K
0
0
20
40
60
80
100
$Price
Product 1
Product 2
Product 3
Product 4
Product 5
Product 6

表格打分：



Product Competitor Analysis

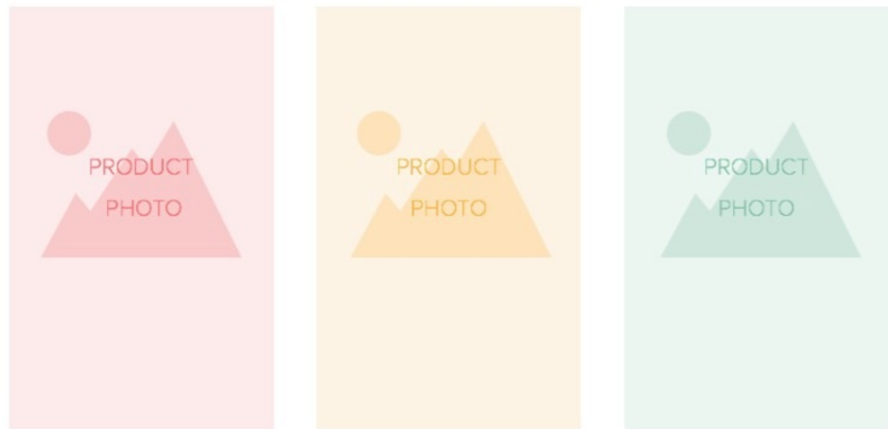
	PRODUCT QUALITY	PRODUCT FIDELITY	AFTER-SALES SERVICE	CUSTOMER SERVICE	PRICE	SHIPPING SPEED
Our Business	92	88	73	66	50	77
Competitor 1	95	84	68	71	55	80
Competitor 2	68	57	78	82	59	64
Competitor 3	85	75	55	64	73	79
Competitor 4	56	58	65	69	90	89
Competitor 5	66	72	82	76	88	67
Competitor 6	71	80	49	52	93	68

持续分析

竞品分析不是一个一次性的分析，它需要融入在产品生命周期去，每当我们遇到困难走不下去的时候，竞品分析能帮助我们找到一些灵感，切记不要直接照搬。

最后，这个网站可以帮助你制作一些漂亮的分析报告:点这里

Product



Strengths & Weaknesses

Strengths
- What makes this service a top tool for addressing consumer needs?
- What are some cutting-edge features?
- Any novel capabilities that sets this company apart?

Weaknesses
- What aspects of this company have room for growth?
- What part(s) could be improved to strengthen services?
- What elements have prompted feedback?

Strengths
- What makes this service a top tool for addressing consumer needs?
- What are some cutting-edge features?
- Any novel capabilities that sets this company apart?

Weaknesses
- What aspects of this company have room for growth?
- What part(s) could be improved to strengthen services?
- What elements have prompted feedback?

Strengths
- What makes this service a top tool for addressing consumer needs?
- What are some cutting-edge features?
- Any novel capabilities that sets this company apart?

Weaknesses
- What aspects of this company have room for growth?
- What part(s) could be improved to strengthen services?
- What elements have prompted feedback?

Market Breakdown



用户访谈

用户访谈是什么

用户访谈是用户调研的一种方式，是最感性的方式，也是最有效了解用户的方式。 为什么最感性？因为是人与人之间的传递，无法避免大量的主观意识；为什么最有效？因为是人与人之间的传递，能感受的纬度最丰富：用户的情绪、状态、偏好、诉求，我们会被非常立体的感知所包围。就如同我们为什么看电影喜爱VR胜过3D，喜爱3D胜过2D，因为我们追求的永远是更真实的感受，而更真实的感受能带你对事物产生更准确地理解。

用户访谈从字面理解很简单，是指采访产品的终端用户，而实际上用户访谈所要达成的目标并不简单，包括用户需求洞察、用户行为分析、用户感知获取等等。用户观察和访谈是渗入到产品的整个周期的：产品初期的用户需求洞察--原型阶段的用户测试--上线前的可用性测试--上线后的反馈收集分析--迭代中的用户数据分析。



我们今天所谈论的重点是产品初期如何从用户访谈中洞察用户需求、要做哪些事情、如何利用访谈结论。

用户访谈的价值：

- 让产品设计者代表用户发声，培养同理心。用户访谈是一个传输“脑电波”的过程，当访谈结束，你就必须让用户的所听、所看、所想传输到自己的大脑里，让自己作为代表为他们设计。就像一个演员，他所演的不是他自己，而是代表着社会上某一类人群，他必须深刻了解这类人。假如在采访阶段没有对用户产生深刻的理解，那么设计出来的产品也必定不能帮他们解决痛点，只是为自己而设计。
- 从了解用户到理解用户，再到摆脱用户。用户访谈让我们充分地、全方位地理解用户，然后我们才能很好地摆脱用户，why？如果我们停留在理解用户的层面，那为什么不让用户自己设计产品呢，我们需要站在比用户更高的地方去看待问题。有时候用户产生的想法和行为连自己都不能理解，例如为什么大多数人吃甜食时有一种幸福的感觉,因为我们

虽然生活在丰衣足食的社会，但是我们的基因却记得远古时代风餐露宿的生活，那时找到成熟的、糖分高的果实是很难的，一旦找就会感觉非常幸福。一个普通的用户可能不会了解这些，作为设计师的我们，需要用丰富的知识回答用户行为背后的疑问。

- 让团队对“工具”有很好地认识，既不贬低它，也不抬高它。用户访谈毕竟只是一个工具，它很好，但不是好到可以挽救一个产品；它也有缺陷，但是不可能完全没用，或者像很多人所觉得的：“调研完和我的猜想完全一样”。这个工具只有当你真正去用了，放在合适的场景下才能评价它的好坏，对一个工具的正确认识或许比这个使用工具本身更重要，起码你不会滥用这个工具，然后得出一个否定它的结论。

用户访谈的目的：

对于一个新产品，它能帮你：

- 定义你的潜力用户群
- 找出什么是用户在产品中的诉求
- 找出高价值人在没有使用产品时是如何解决痛点的

对于一个已存在的产品，它能帮你：

- 了解存量用户的行为特征和偏好
- 找到存量用户使用产品时的痛点
- 学习如何留住存量用户，同时获取新用户

如何挑选要访谈的用户

访谈的第一步就是要知道，你要访谈谁、谁是这次的重点、谁不在这次访谈的范围之内。我们应该在采访之前就有一些分类，以下是一些可以参考的用户分类：

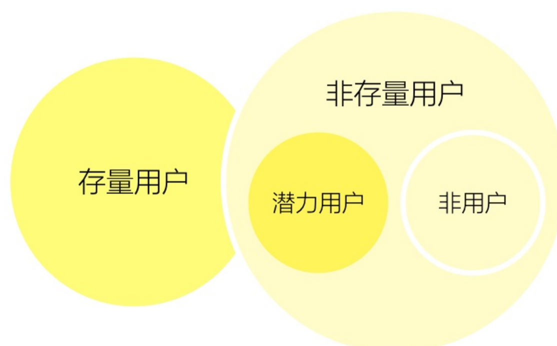
1.可以从用户的类型上挑选：

- 利益相关者：给项目资金和资源的人，能够决定项目的目标和方向；
- 用户 ：是使用产品的人；
- 顾客：是购买产品的人，有的产品顾客等于用户，有的产品顾客不等于用户，比如给儿童用的学习型产品，儿童是用户，但是买产品的是父母，所以父母是顾客；
- 专家：是指在产品领域内的职业玩家，有丰富的经验，比如要做理财产品，那些金融行业的人就是专家；
- 其他相关者：指的是用户周围的人，包括家人、朋友、同学等等，他们能够影响用户使用活购买产品。



2.可以从存量用户和非存量用户挑选：

- 存量用户：如果你的产品已经运行一段时间，那么需要从后台数据中将用户进行分类，如：按照活跃度最高、收入贡献最多来分类，对于这类种子用户需要跟踪他们的行为轨迹，了解他们更多的诉求。
- 非存量用户：指的是现在没有使用产品，但是我们希望囊括进来的用户。如：对于线上支付产品，大家都想让最主流、消费力最足的城市白领成为主要用户，但白领也许对产品并不买单。用户访谈帮我们清楚地意识到哪些非存量用户是有可能被吸引的，哪些不能。非存量用户里面又可以分成潜力用户和非用户，潜力用户是一些使用竞争对手产品的用户，要了解他们为什么偏爱对手；非用户要了解为什么他们不使用产品，有没有可能让他们变成潜力用户。



3.可以根据猜想的纬度挑选用户：

对于一个新的产品，就算没有非常确定的用户类型，也会有大概的用户指向，我们可根据自己猜想，用枚举的方式把用户都列下来,例如：城市白领、二线城市打工族、家庭妇女、老居民...等等。不要害怕过于主观，我们之后可以在访谈的时候再逐渐修正这些分类。

尝试着猜想我的产品会有哪些用户



图像来源于dribbble

值得注意的是：三种挑选的纬度可能是包含关系，如：从存量用户里面挑选出来的用户，又可分为终端用户、顾客、专家，从非存量用户挑选出来的用户，又可分学生、打工族、白领、金领等等。

用户访谈前要准备什么

- 用户招募：提前一周开始招募，人数上需要有冗余，避免有用户临时不来的情况；每类用户至少3个人以上，重点类型招募比例可加大；核对招募用户的有效性，因为招募公司业务专业性不强，找来的人有时不是非常契合。
- 访谈提纲：一般调研团队是由几名设计师组成的，为了统一大家采访的思路，必须要根据每一个采访的对象制定采访提纲，如果带有实地考察的调研，要制定观察提纲。由于提纲是事先做好的，所以不可能完全符合采访的需要，所以在每一次采访过后，大家需要统一的调整提纲的问题，删除不必要的问题或者增加有价值的问题。
- 时间计划：访谈非常耗费体力，需要有好的精神状态，一天访谈不要超过6个，一个访谈不要超过1小时，1小时以上的提问已经得不到可信的答案。
- 人员分工：两人一组，一个人提问，一个人记录，为了让每个角色都有更好地参与，可在访谈中交换角色。如果有可能的话，尽量让决策者、产品经理、开发也参与前期的调研。
- 奖品&资源消耗：准备金额50元-200元的奖品（有一些吸引力，但也不至于花费太高），提高被访者的参积极性；准备一间温馨的办公室/studio，营造轻松自然的氛围；如果是外出访谈，准备一些易拉宝、工作牌之类的，表明身份的真实性。

几种访谈的方式

1.一对一访谈：

我们用的最多的是一对一访谈。我们要分清楚，针对哪类用户用什么方式最有效:

对于 **To C** 的产品，可采用坐谈的方式：

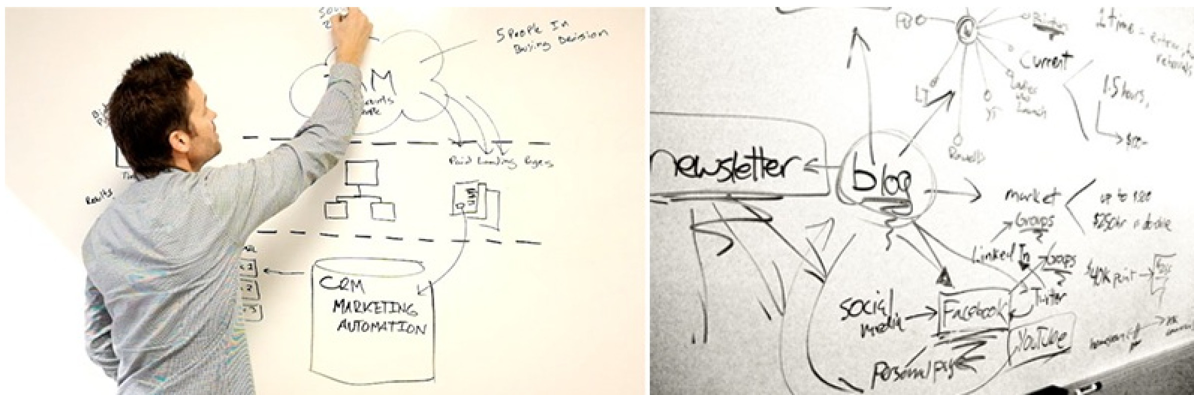
To C的产品一般都是一个角色贯穿产品的始终，不用和其他角色配合完成某项工作，所以我们需要深入了解单个用户的方方面面，如：行为习惯、偏好、生活场景等。采用坐谈的方式帮助我们由粗到细提问，再针对某个问题分支深入挖掘。

对于决策者的访谈，同样应该采用坐谈的方式，因为领导时间少、意见重要，一对一可以问得很深入，并让他感到足够的重视。



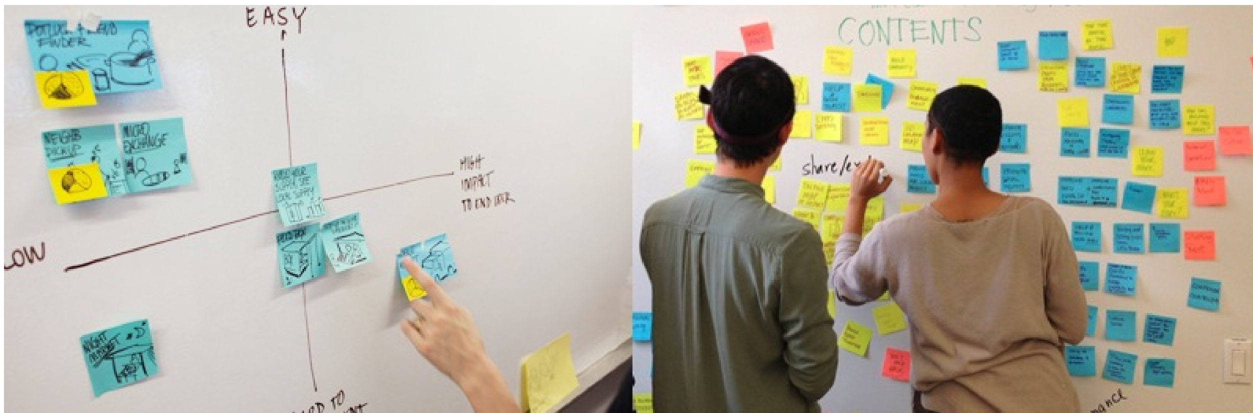
对于 **to B** 的产品,可采用可视化访谈的方式：

- To B的产品，用户通常都会有上下游关系，上游把工作做完传到下游。我们需要了解的是：业务流程、人员配合关系、每个阶段的产出等等。可采用可视化的方式，如下图，边采访边画出整个工作流，再了解每个流程中的问题。



2.Workshop访谈

针对企业项目的产品经理，我们通常采用workshop的形式，因为产品经理的人数比较多，大家有各自的想法不容易达成一致。workshop的好处就是能让彼此了解想法并且意见达成一致。



3.电话访谈：

电话访谈是特别需要技巧的访谈，想象一下我们平时接到陌生来电的情景：

1. 看到陌生电话眉头一皱：谁啊，骗纸吧
2. 接起电话不怀好意：请问你是？
3. 还没等对方讲完一句话，就说：不好意思不感兴趣再见，啪。



所以电话访谈首要解决的事情是要消除受访者的疑虑，否则你将一无所获。如何解决呢？

- 需要用一个“官方”号码拨打电话来消除疑虑；
- 通过受访者熟悉的平台邀请参与调研（如官网、facebook等），并且把时间、所需时长、主题告知受访者，让他有充足的准备；
- 赠送小礼品，让受访者乐于表达。

第二要注意的事是尽量直击要害，要重要的问题放在前面询问。因为你和受访者无法面对面，你不知道他接电话的时候在做什么，是否是一心二用或者心不在焉，所以在得到信任以后需要立即问一些核心的问题。对方不愿意回答的问题不要强迫追问，造成受访者的方反

感，最后，如果发现受访者开始不耐烦，就快速结束对话，因为他提供的信息参考价值降低。

4.实地考察：

实地考察的目的是为了从更客观的角度观察【事情】发生的进过，这里的【事情】可以是产品使用的流程，可以是用户痛点的产生和消解，可以是设计机会的若影若现，我们通常所用的方法是“守株待兔”和“主动出击”。守株待兔是对事先制定好的观察目标进行观察，对用户产生情感的认知，对过程提出问题；主动进攻是对问题进行回答，从各种角色、各种角度中获得答案。

访谈提纲包括的纬度（每个产品不同，问题不同，仅供参考）

可以分为几类问题：

- 基本信息：年龄、职业、工作情况居住地等
- 过程或经历：和产品相关的几个关键的场景，例如存款取款场景、买理财产品的场景等
- 态度动机：产生行为的动机
- 痛点：根据场景挖掘痛点，深入访谈
- 尝试过的方案：曾经尝试解决痛点的方案
- 愿景：希望如何改进痛点
- 工具技术：平时解决此痛点用到的工具有哪些
- 解决痛点成功的标准

一些问问题的小技巧：

- 介绍访谈的目的和所需时间
- 把一个问题控制在20个字以内
- 一次只问一个问题
- 问题尽量贴近用户的真实场景
- 将问题集中在用户的行为轨迹上，而非感受上
- 提供一些日期的提示帮助用户进行回忆
- 问题中带有专业性的，转化成用户的语言去问
- 使用中性的词或者语句，不让用户产生偏向
- 在必要时才涉及用户的感受或者隐私问题
- 问完一个问题时，给用户一些思考的时间，不让着急让他回答

一些访谈时避免做的事情：

- 问用户“你想要什么”
- 让用户做选择，而选项并不符合他们实际的情况
- 问一些引导性强的问题
- 讲问题放置在错误的前提下

- 让用户讨论TA没有经历过的事情
- 让用户预测将来
- 用过于专业的词汇问问题
- 用过于情感化的词汇
- 处于好奇心问一些私人问题

相互制约的方法

每一个方法都不能单一地存在，必须有另外一个方法去制约和测试它的真实性，用户访谈是一种定性的方法，可以用如问卷调查、数据研究这样定量的方式来验证访谈的有效性，将用户访谈的结果导入到数据或者市场调研中去做测试，把有差异的地方拿出来做讨论，反复验证才能得到更接近真实的答案。

用户画像

在做用户体验设计时，最常挂在我们嘴边的词之一就是“用户”，而这个被用烂了的词在设计中并没有起到良好的作用。

产品经理和设计师虽然都在说“用户”，但事实上，他们可能并不知道真正的用户是哪些人、他们有什么特征。因为大多数人没有做真正的用户调研，从而无法培养同理心而把他们自己变成目标用户（除非他们真的是目标用户）；而就算前期做过用户调研，他们也很难记住用户类型到底是哪些，到头来，还是按照自己臆想的用户来设计产品。

了解和分析用户最好的办法就是建立persona（也可以叫建立角色或者用户画像），它是产品最重要的设计工具和沟通工具。

persona从何而来

persona一词最早出现在2004年出版的《The Inmate Are Runing the Aslyum》中，由交互设计之父Alan Cooper所创造。

要做一个能够称之为“好体验”的产品，必须对用户有足够的了解，从而建立同理心——即通过和用户的沟通、观察他们的生活建立起一种和他们感同身受的心理状态。persona就是设计师对用户的观察、与用户沟通后分解出来的几种人物类型，每一个类型称之为一个persona。

persona为什么重要

很多公司的老板或产品经理喜欢做“大而全”的东西，动不动就说要做一个XX平台、铺全量用户。这种想法使得他们在产品中加入了很多不符合真实需求的功能，例如：造一辆车需要同时满足孕妇、驴友和大学生的需要，孕妇想要更宽的座位来平稳放置孩子，驴友需要有坚实的底盘穿越各种地势，大学生需要有拉风的敞篷，如果要满足这三方的要求，设计的车将会是这样：



可以看出，这是一个“三不像”的设计，并不能很好满足任何一方的需求。如果我们想做最好的体验，当然是为这三种用户都分别造一辆车，但是现实中并没有这么多的成本让我们这么做。所以当我们只能做一个产品而服务于多种人群时，最好的办法就是让其中某类用户的体验做到100分，这群人虽量小，但是只要让他们狂热的喜欢，就会有强大的裙带效应。反之，如果把所有用户的需求都做均衡的考量，那么所有人的体验只能都维持在70分左右，我们把这种设计叫做“无任何满意度设计”。

persona就是【基于定性与定量研究】来帮助你精准定义某类用户做“100%满意设计”的工具。

何时建立persona

persona一般都是在一个项目的最开始，在访谈用户、收集用户数据之后，用persona 来收敛所有的用户资料。

如何建立persona

1.将采访用户分类

根据访谈提纲，将所有采访来的资料结构化输出，应包括至少三方面的信息

- 基础信息，如：年龄、性别、居住地等；
- 行为的变量，如：财力、对互联网的接受程度、对产品的依赖程度、受教育程度、用户使用产品的频率等；
- 用户的三层目标：本能目标、行为目标、反思目标。本能目标是指用户对视觉层面的偏好和感觉，影响产品的视觉设计；行为目标是指用户做某件事情所想达成的目标，是用户的心里模型，影响产品的交互设计；反思目标是用户在使用产品之后，想要获得的生活和精神上发生的改变，影响产品"使命感"和“意义”的设计。



对于一些无法结构化的信息，可以先保留下来作为参考，如果觉得非常有价值，可以把它标注出来。

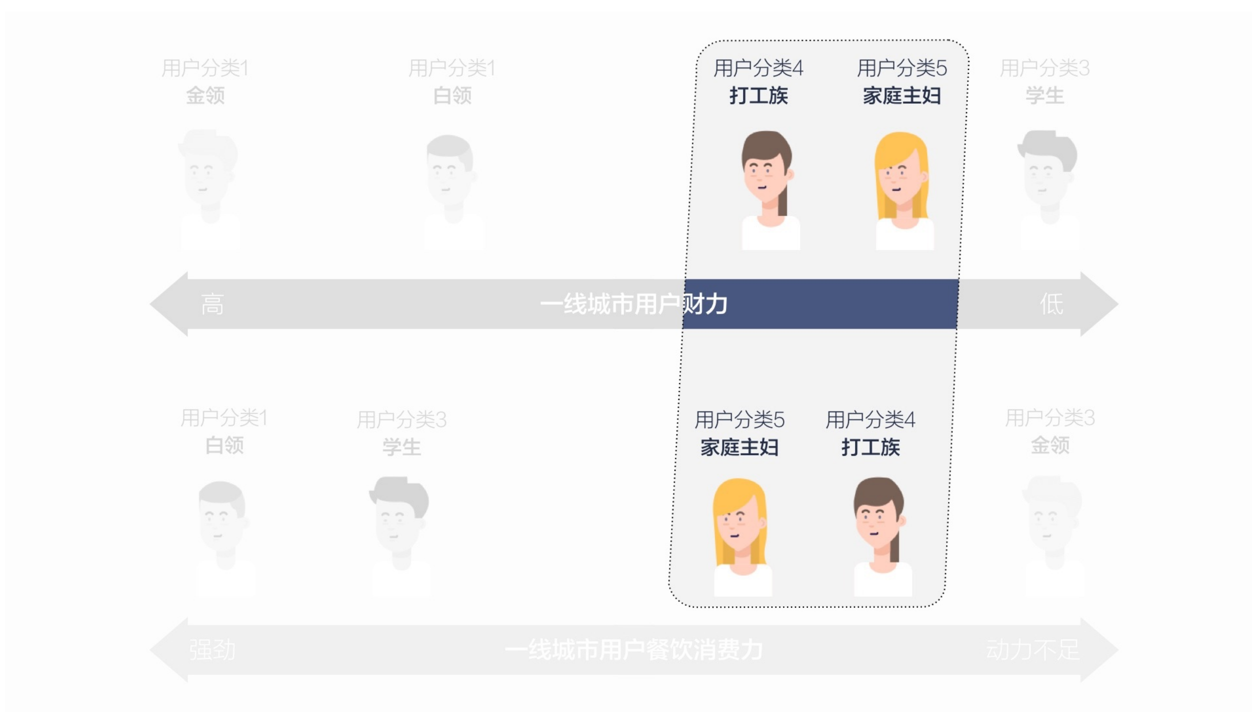
2.将行为变量进行比较，把相似类型合并

- 将我们发现的行为变量进行横向比较，并不用非常精确地用数据对比，只用判断每类用户在轴线上的先用关系即可：



- 当我们把用户比较之后，可能会发现有几类用户是很相似的，这时需要我们判断，是否把它们合并成一类。合并的依据可以是：关键信息很相似或类型过多（多余的用户画像会干扰产品做判断）。

> 如下图所示，这两类用户在财力和消费力上都有很强的趋同性，并且这两个轴线是产品的关键指标，则可以合并两类用户，尽管他们可能在其他方面有较大不同。



- 合并之后，我们需要总结一些行为模式，如：收入高的人群消费力也强、有了孩子的人

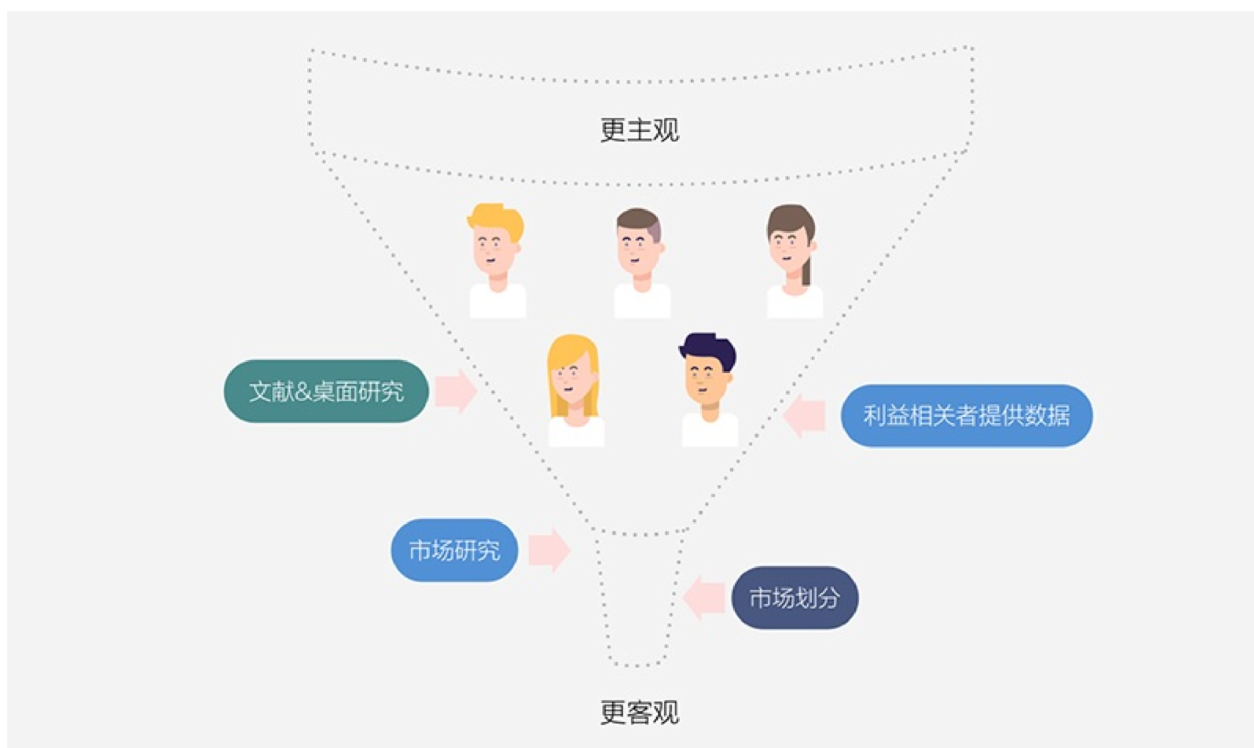
用钱更谨慎。

3.用一些市场数据来验证用户画像的真实性

采访用户和用户分类都带有一定的主观性，所以需要收集一些市场数据来佐证用户画像的真实性。例如：从采访的用户当中发现："一线城市白领大多不如二线城市白领的消费力强劲"，这类"反直觉"的结论需要从市场分析报告中找到佐证，如果不符合调查报告的结论，就要对此做推测，为什么在此产品上，会有这样的现象产生，是否有特殊原因等等。把市场调研报告中的一些数据、竞争产品的用户数据加入到分类当中，能让用户分类在客观性上有数据支撑。

我们应收集的数据有：

- 市场研究数据
- 市场划分模型
- 文献&桌面研究数据
- 利益相关者提供的数据



4.制作具体的画像，将数据用图标可视化出来进行对比

- 用【一个】具体的人来代表某一分类用户，这个人需要非常具体化，有真实的照片、姓名、年龄、性格特征、财务状况等等，把结构化信息中最重要的几个纬度摘出来，成为一个完成的persona；
- 用雷达图、曲线图、柱状图等图标将一些可数据化的信息展现出来，并且给persona打上一些标签，如："消费力强劲的白领"；

5.展开行为描述，使用辅助工具

第四步的信息能让人很快了解一个persona代表的是哪类群体，他们最显著的特征是什么，而要把这个persona完完全全地展示出来，就需要一些更具体的叙述，可以用以下方式进行描述：

- 从采访中提炼的一个生动的故事，可以带有一些虚构的成份；
- 一些和persona相关性的拼贴画，表现TA的所看所听所想所讲；

6.指定主**persona**

在详细了解了各个persona之后，我们需要选定1-2类作为我们的设计目标人群，并未他们排列优先级，可以用以下纬度来排列：

- 首要persona
- 次要persona
- 补充persona



7.将其他角色的目标写出来

以下这些人群的的目标，也会作为产品设计的参考方向：

- 顾客目标：顾客在花钱时，想要达到的是什么目标，如安全目标、愉悦目标、教育目标等。
- 决策者目标：决策者做这个产品，最终为了什么目标，如利润、市场占有率、资源利用率等；
- 技术目标：技术人员在实现产品时需要保证的目标：如安全性、数据真实性、跨平台统一性等。

persona的好处

1.与所有人目标一致：

如果你是乙方，只要甲方客户认可了这个persona，就可以避免他们以自己的喜好来判断设计，让他们清楚我们并不是为他们设计，而是在为用户设计。在功能入口上、架构交互上、视觉上都首先满足主persona的需要和喜好。

值得注意的是，我们虽然优先满足主persona的需求，但并不是一味损害其他用户的体验，主persona需要达到“良好而惊喜的体验”，次要的persona需求也应确保“能顺畅使用”。

2.作为沟通工具也十分有效：

在没有persona之前，程序员、设计师、产品经理很容易把用户作为任何一个普通人去考虑，甚至很多时候是从自己的需求出发，但是事实上他们并不是目标用户，目标用户也通常不是随随便便一个普通人，他们有自己的特点和需求。

我们必须把调研得出的persona引入设计中，而不是做完之后把它丢在一边。例如在我们一个国外项目中，主persona是一个叫做sofia的女分析师，大家要谈论用户的时候，从来不直接说“用户”，而是以“sofia”来代替。这样做的好处是让团队里的每一个成员，不得不对角色的特征了然于心，所有人必须像了解你身边的一个朋友一样了解persona，这一点非常重要。

3.提高设计验证效率：

当我们把低保证原型做出来之后，可根据persona快速验证方案，为设计师提供了客观的优化依据。虽然这种方式比不上真实的用户测试，但能更快速更低成本。

值得注意的事

用户画像不等于市场的人群划分

在我们做调研的过程中，常分不清楚用户调研与市场调研的关系，因为大家好像都要基础用户做分类。其实他们最大的区别在于用户画像是以用户行为和动机分类，而市场的人群划分则是根据统计数据、购买行为来分类。很少会出现市场划分人群和用户画像一一对应的关系。但是市场分析缺可以作为过滤器来锁定我们索要采访和分类的人群范围；而用户画像和可以为市场的人群分类提供方向性的建议。

用户体验地图

我曾看到很多设计师设计技法非常娴熟，能用ps画出超写实的图标，把app设计得花枝乱蹿，在dribbble上混得风生水起。但是当他去show设计时，却常常败下阵来，原因有很多：不了解真实用户、没有数据的支持、不清楚产品目标、经验不足等等。总的来说：无法说一个好故事。

本篇要讲的用户体验地图，就是帮助设计师讲故事的工具。

端到端的信息图

下面用这张图精确地描述用户体验地图包括什么内容：

Rail Europe Experience Map

Guiding Principles

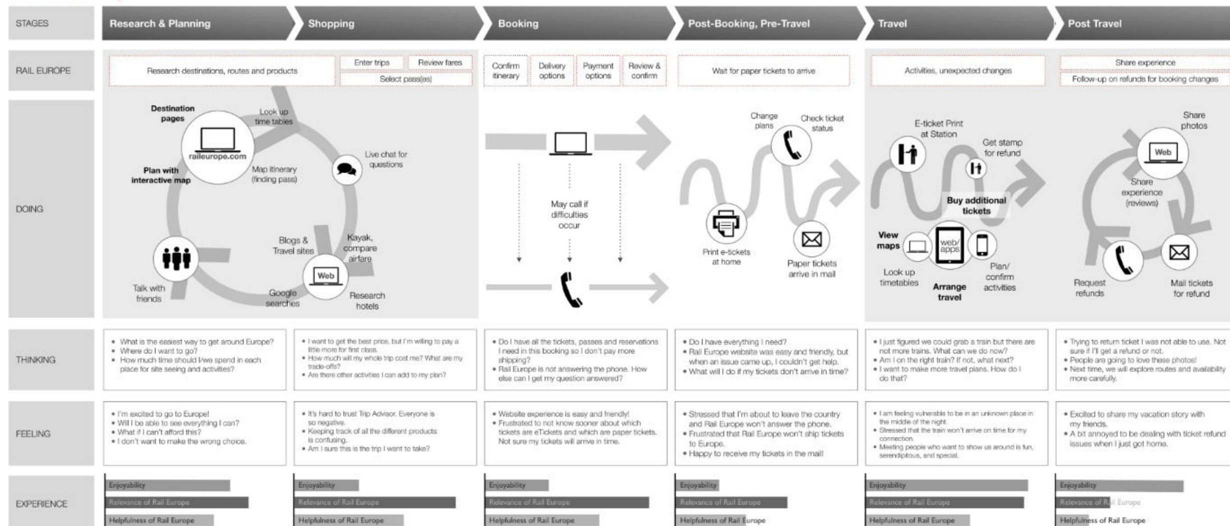
People choose rail travel because it is convenient, easy, and flexible.

Rail booking is only one part of people's larger travel process.

People build their travel plans over time.

People value service that is respectful, effective and personable.

Customer Journey



Opportunities



Information sources: Stakeholder interviews, Cognitive walkthroughs, Customer Experience Survey, Existing Rail Europe Documentation

adaptive path

Experience Map for Rail Europe | August 2011

可见，用户体验地图是一个呈现【端到端用户体验流程的视觉化信息图】，以用户视角为出发，用视觉化的语言来描述：

- 一个好的故事（用户所看、所听、所想...）
- 一个支撑产品的后台系统
- 新的设计机会

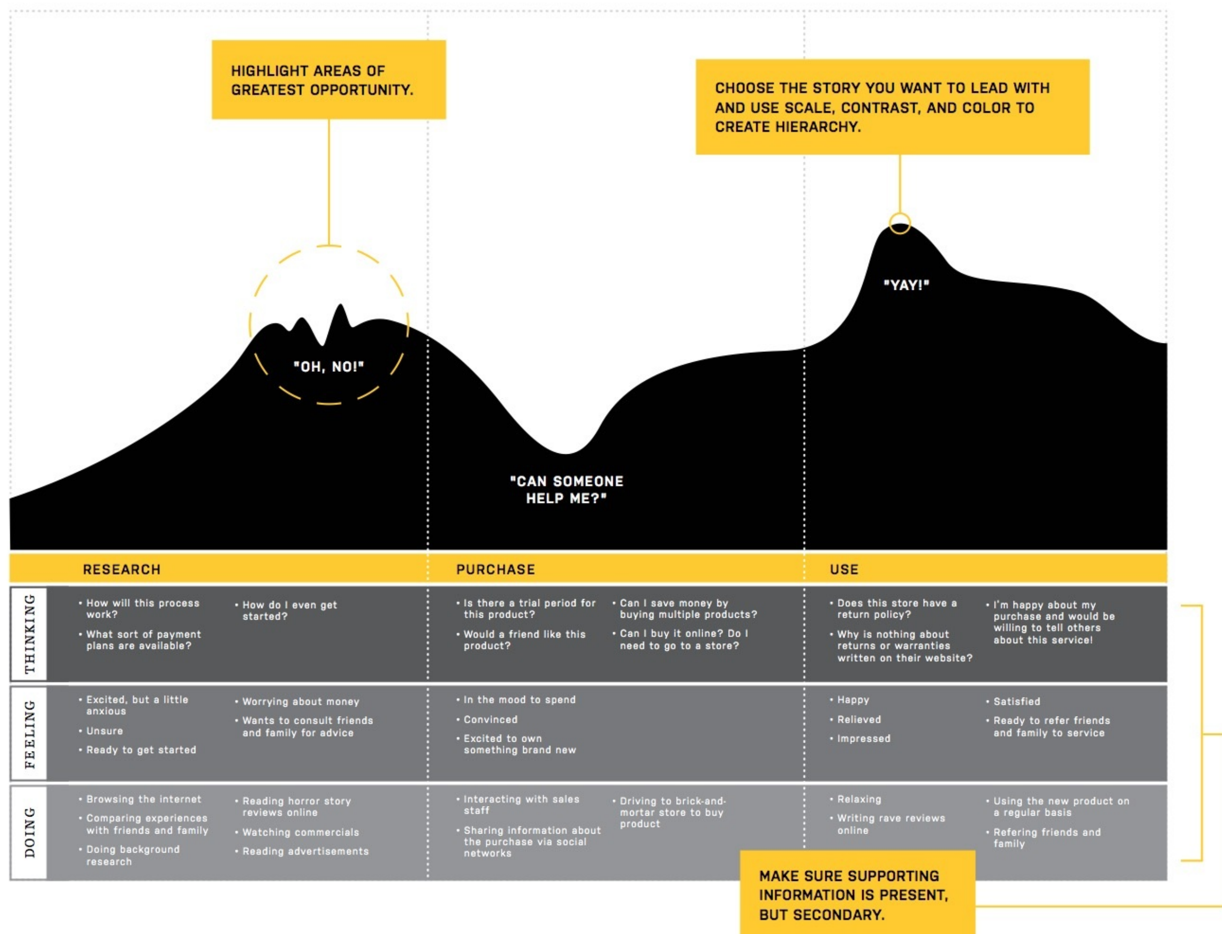
- 新的商业价值

1.一个好的故事

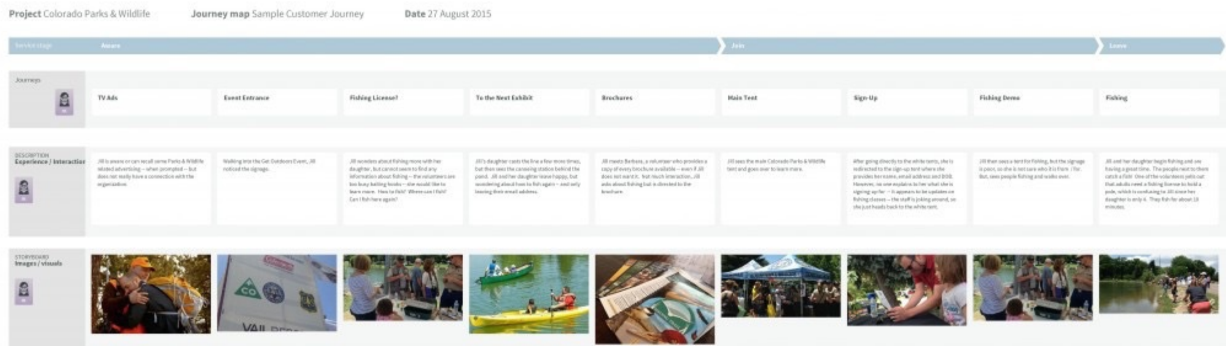
用户体验地图通常和persona在一起使用，共同讲一个生动的故事。persona是这个故事的主人翁，而用户体验地图负责展现这个故事的开始、过程、高潮、结尾，表达用户在流程中所展现出来的态度和情绪。和persona一样，帮助梳理用户调研的信息、使内部利益相关者基于共同的知识背景讨论问题。如果persona是一个点，那用户体验地图就是一条线。

故事的内容：

- 用户旅程的每个阶段
- 用户每一步索要达到的目的
- 用户每一步的情绪态度



我们可以用【连环画】或者【照片】的形式来表达用户旅程的阶段，有很强的代入感。



2.一个支撑产品的后台系统

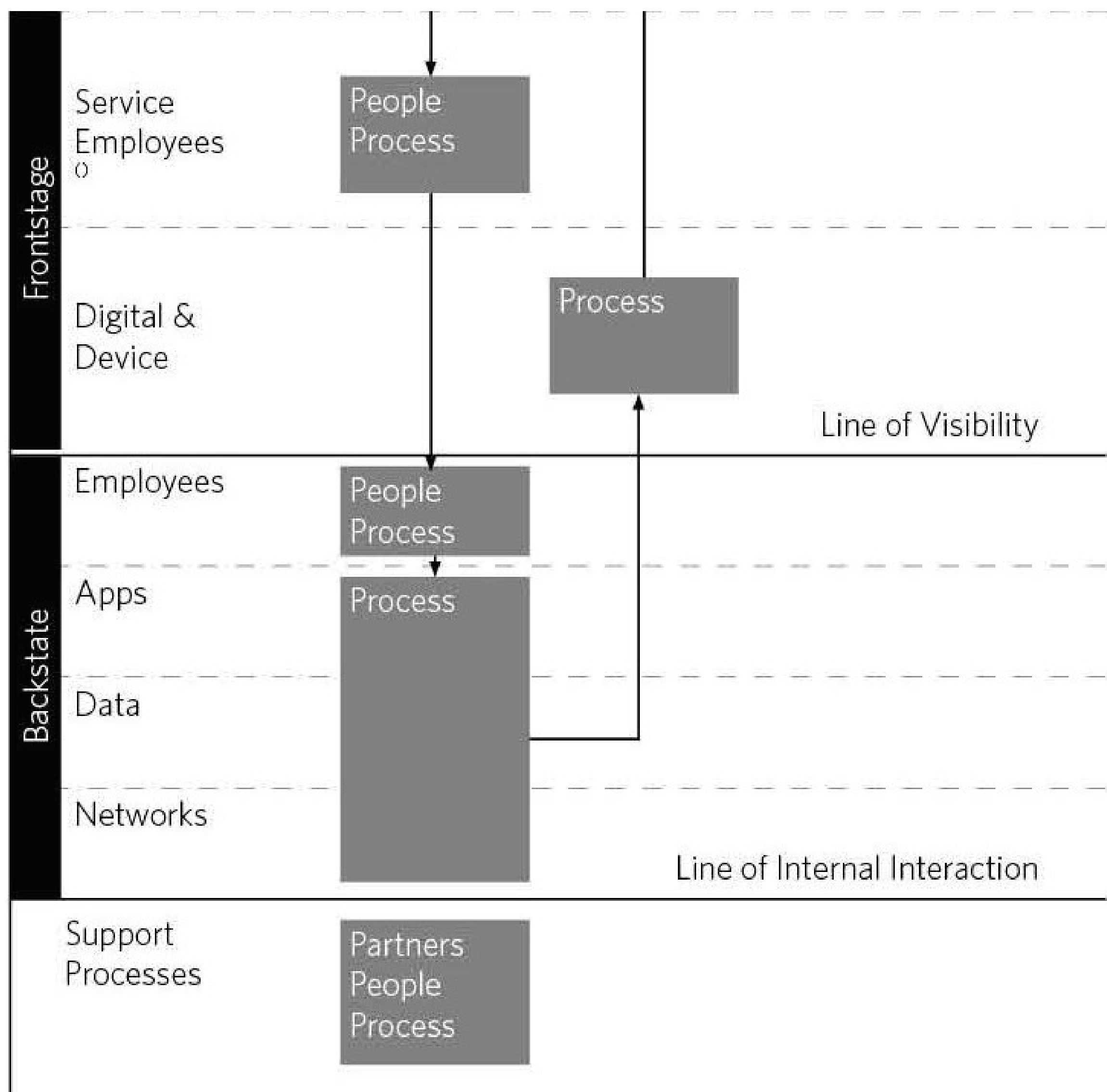
如果作为一个新的产品，我们不用太在意后台系统，因为新产品需要大量的试错，太早考虑后台系统会浪费很多资源。但如果是一个已经成熟的、需要持续运营的产品，后台系统将会对企业内部起着至关重要的作用。就像搭一座桥，好的故事是桥面，好的后台系统是桥柱子，支撑桥面正常使用。 在腾讯工作时，一些to C的产品通常是没有后台系统的，所以当我们进行一些数据收集的时候，需要开发人员大量导出数据，再靠运营同事手动加工成报表，从提出需要到拿到数据再到视觉化呈现，需要好几天的时间。这种方式很大程度浪费了资源和时间，且有一定的出错率，对于一个可持续产品来说效率很低。

所以，当我们面对一个可持续运营的产品时，就必须把它的后台系统考虑进去。例如，大众点评是一个2C的产品，大家看得到的是一个漂亮的app，看不到的是背后有一个强大的商家门户平台，提供入口让商家上传门店信息、图片、运营活动等。

我们希望将后台系统尽早地考虑在user journey map里面，帮助预估产品的资源和工作量。

后台系统内容：

- 与用户发生交互的企业内部角色，如销售、电话中心等
- 与用户发生交互的企业内部系统，如email、网站、app、电话等
- 每一步所涉及的后台系统的所需要的操作步骤，如：创建用户、发送email、发送提醒、打电话、更新信息等
- 使用后台系统的内部角色：IT、运营、财务等
- 需要关联的系统：邮件系统、CRM、SAP等



可见内部系统是一个比C端产品复杂得多的产品，它更多考验的是产品经理、设计师的逻辑能力和需求梳理能力，做好系统能够建立一个良好的企业生态系统，释放如财务、运营、接线员这样的人力。

3.新的设计机会

这里的设计机指的是我们针对体验地图中用户情绪、心态比较负面的情况，提出的改善建议。同时，考虑到改善建议背后所需要的资源，让建议得到合理地评估。

新设计内容：

- 每一步能够挖掘的设计机会点
- 每一步想要影响用户情绪、行为产生的改变
- 每一步所需的资源（人、资金、物资）
- 试着画出新的流程，把用户行为放进去，看看是否能产生一些好的变化
- 试着用一句话来表达你想要做的体验提升：如果我们创造...将解决什么问题...做这些需要...结果是....

4.新的商业价值

作为商业产品，一切将会以“钱”来衡量最终的成效，所以，如果你的建议能够直接指向开源节流、新广告盈利的话，就能帮企业创造价值。

案例：XXXX是一个为如家、莫泰提供wifi硬件设备和软件服务的公司，公司内部的呼叫中心为这些快捷酒店解决软硬件故障问题。通过调查发现，来电报修的远程关单率为90%，意思是：90%的保修，都能通过接线员远程指导来解决，而且很多问题的解决手法又都是类似的，如重启、未连接XX系统、账号错误等。所以，我们提供了一个解决方案是“建立报错知识库”：让店长、店员在打电话之前，通过选择题对问题进行定位，系统自动提供解决方案，从而减少来电保修数量。这个公司每天的接电话率大概在1000个左右，如果能让远程关单率从90%下降到80%，就能减少100个来电。接线员每人每天的接电话个数是45-50，每天减少100个电话，就相当于释放2个人力。

我们在提出建议时，应考虑到背后的商业价值，让它成为老板改进产品的动力。如：

- 呼叫中心的电话减少数量X单次通话的费用＝每个月/年减少话费
- 新增用户X每个用户平均的利益贡献度＝增加的收入
- 新增产品的广告次数x广告的单价＝增加的收入

如何制作用户体验地图

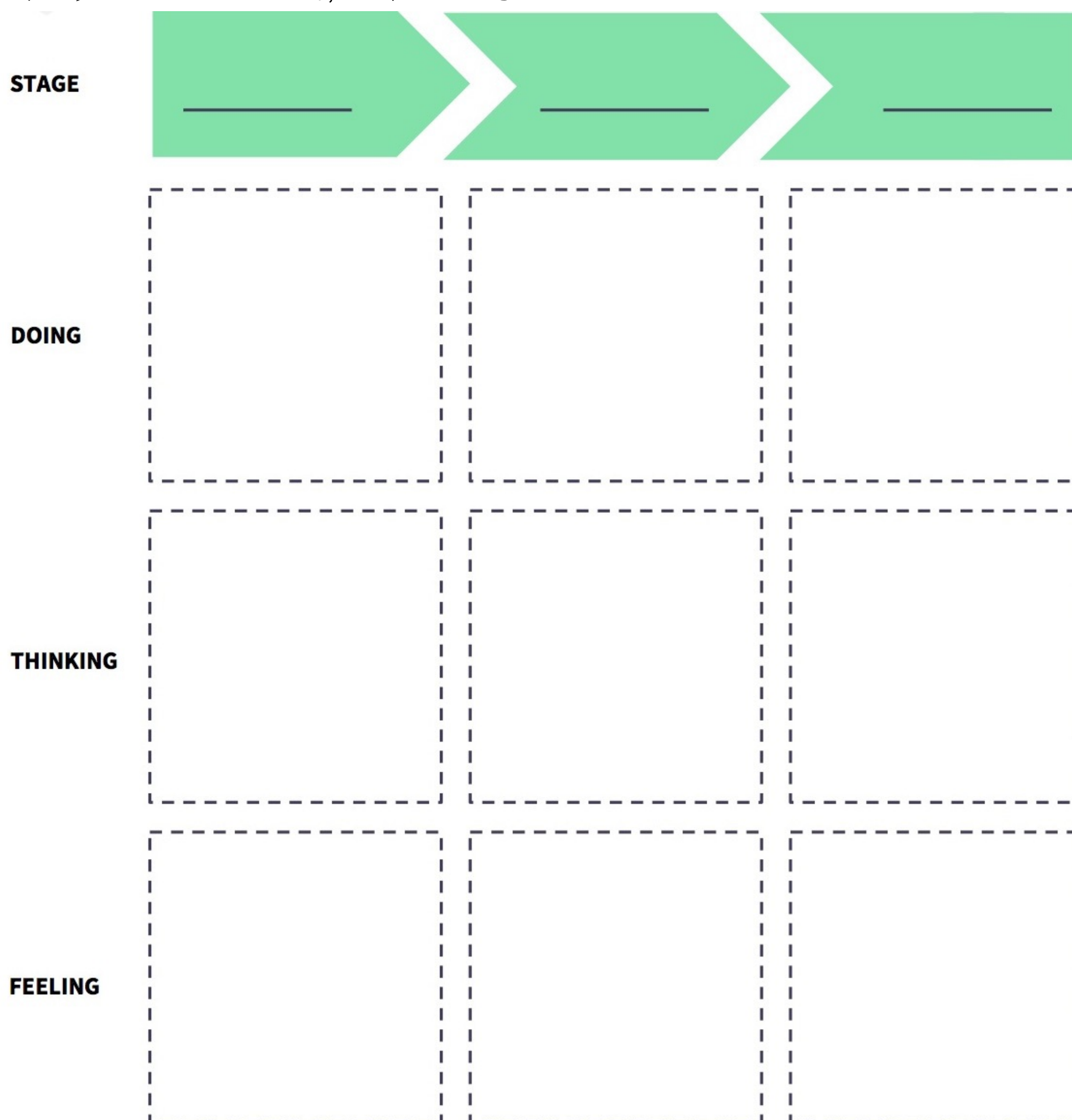
制作用户体验地图的方式和任何做设计的方式一样：猜想-验证-实施。

1.mini user journey map workshop

在进行进一步地调研之前，这个workshop的产出物是一个简版的user journey map，是基于产品经理、设计师、市场人员自己的经验和知识背景得出的。需要基于之前得出的persona和产品所产生的数据，这一过程更倾向于定量。这样一是能跟验证阶段的产出成果进行很好地对比，二是能让我们更有针对性地挖掘问题。

步骤流程：

1.整理用户步骤：这里的颗粒度不必太细，但是要把行为的阶段性体现出来。 2.加入态度：在每一步标注出他们的态度,如:高兴、无感、低落



2.探索验证

这一步所要做的事情，是拿着之前的mini user journey map 放到真实的场景中去检验，所采用的方法可参考用户访谈，只不过更倾向于把流程中的每一步展开来进行更深入的调研，攫取用户每一步的感受、目标。可以采用但不限于以下几种方式：

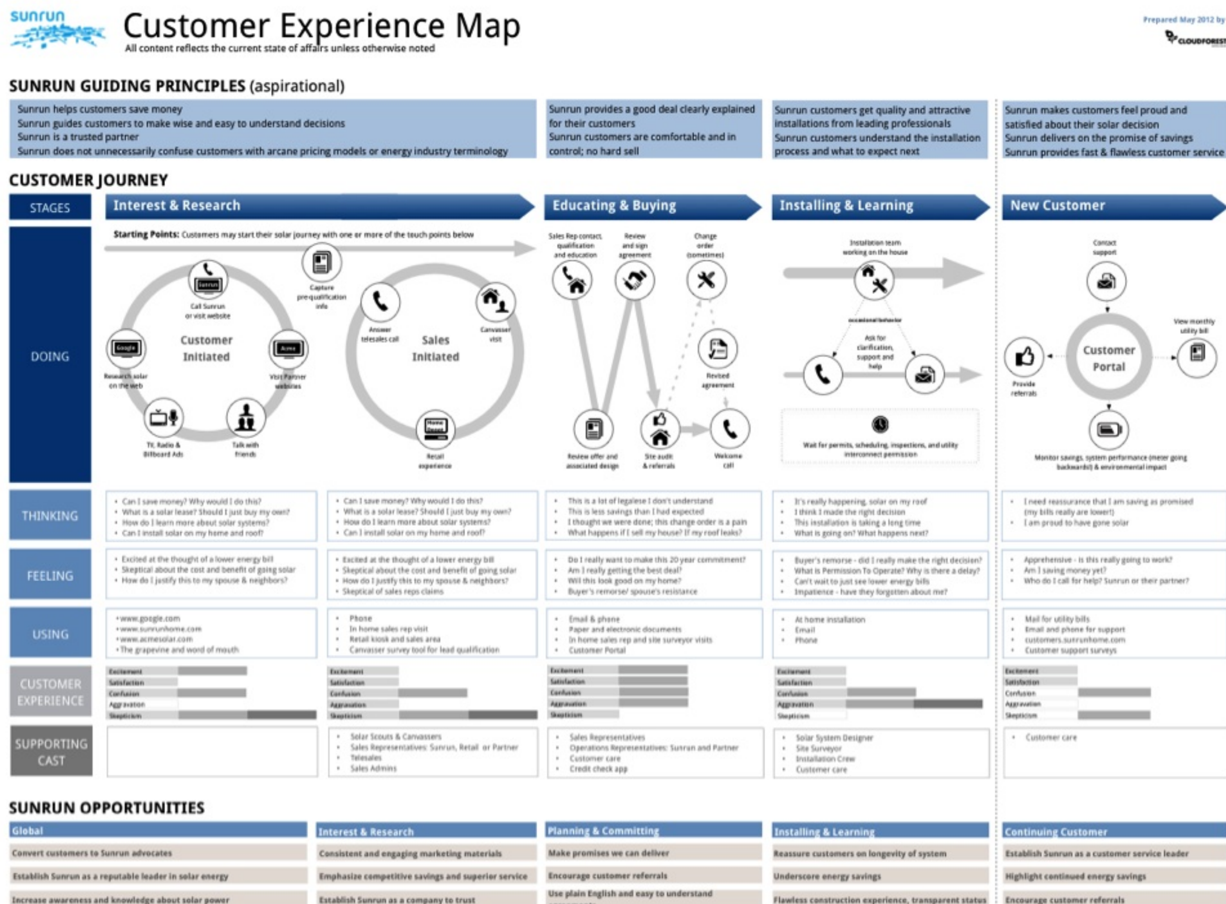
- 电话访谈

- 焦点交租
- 用户日志
- 面谈

3.用户旅程图workshop

当验证完猜想之后，需要组织一次workshop，这次需要全部利益相关者一起构建完整流程图、同时达到同步信息、发现项目机会点的目的。

这里就不做赘述，把之前的信息集合起来，为大家展示一张完整的用户体验地图：

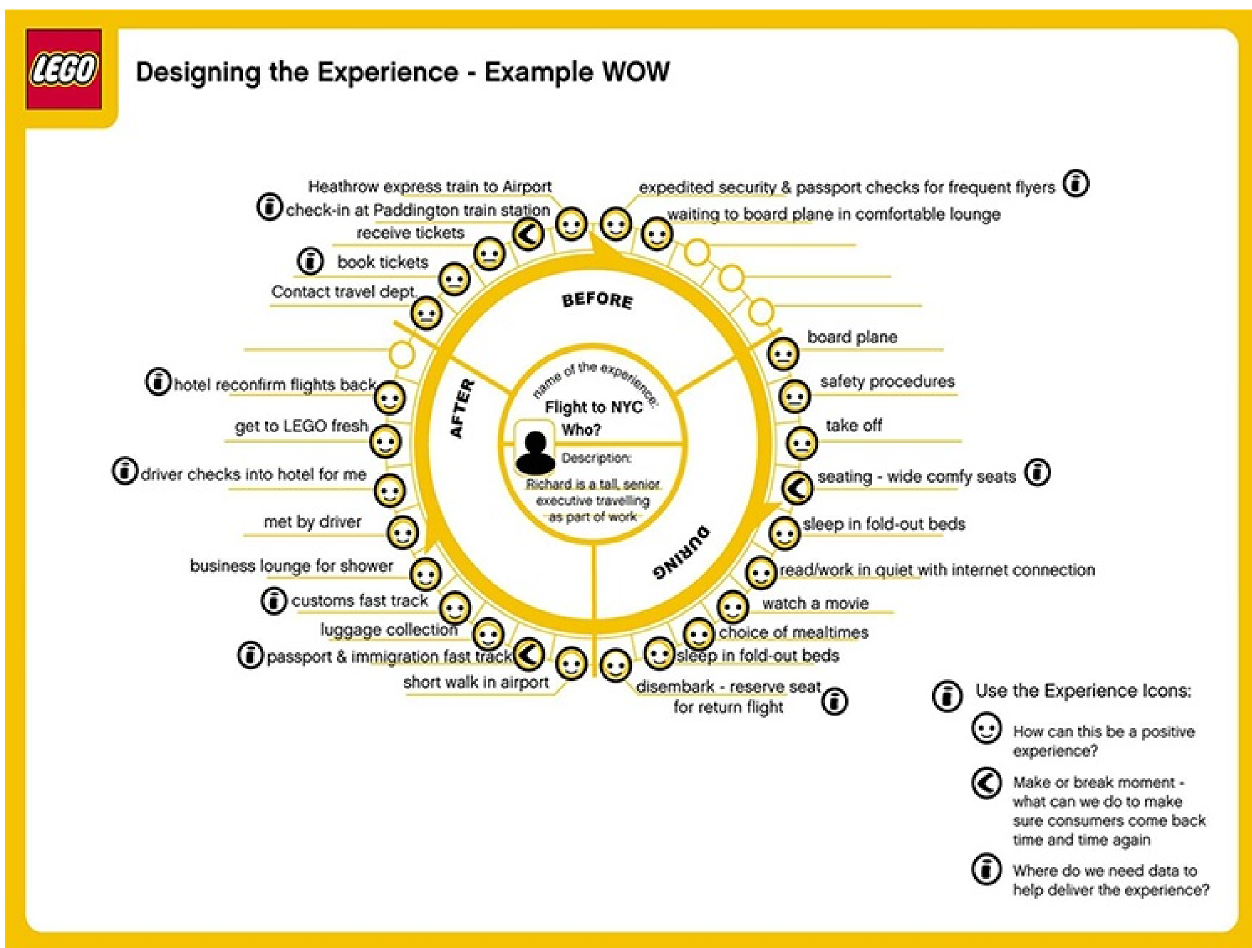


几种地图介绍

上面所介绍的是其中一种比较通用的用户体验地图，接下来介绍5种其他不同类型的用户体验地图。

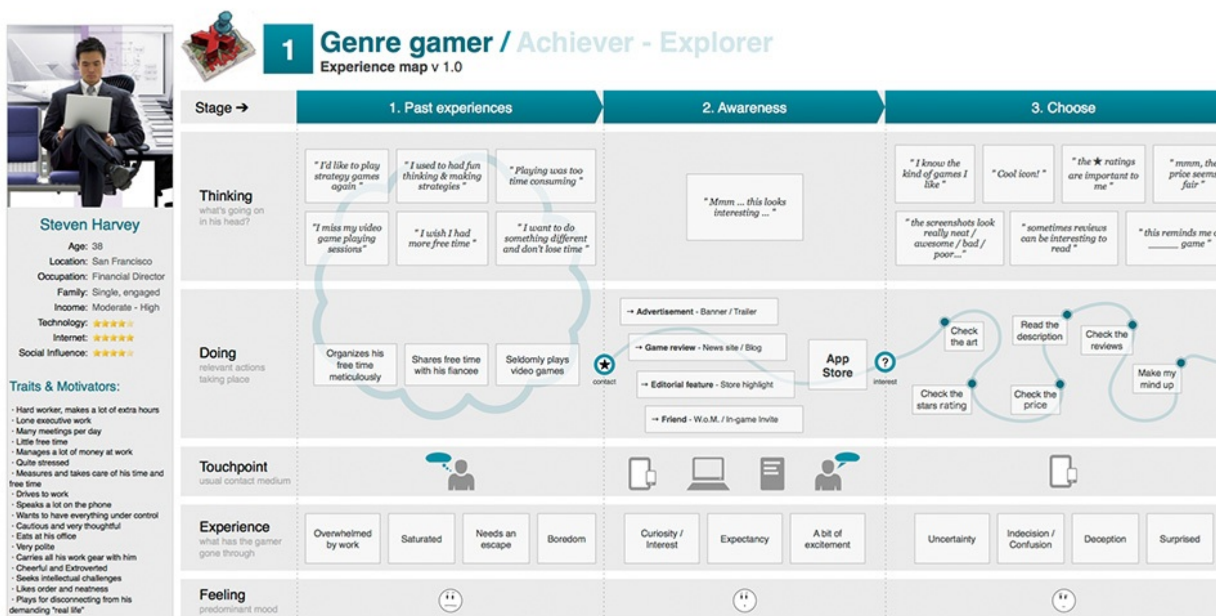
1.轮轴图

这是一个乘飞机去纽约的用户体验地图。中间圆用来描述persona，外面的圆被分为三个步骤分别描述出发前、旅程中、达到后每一步的用户行为和情绪。这类型的图简单高效，但是不容易把问题描述得全面，适用于业务简单的产品。



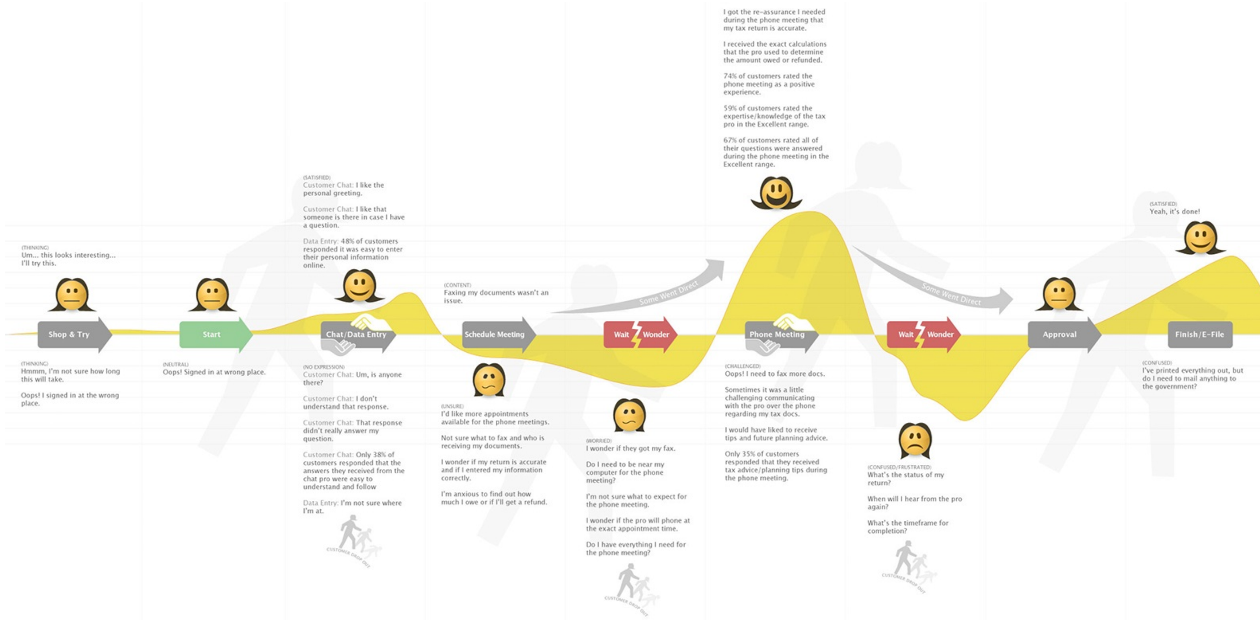
2.标准体验地图

这是一个标准的用户体验地图，讲的是一个金融经理如何通过产品改变了之前的行为意识，以及产品如何影响他做决策。这张图里面最关注的是人的行为、目标、习惯、动机，帮助我们决定哪些体验流程是我们想要改变的。



3.典型客户旅程图

这类地图关注的是客户（买单的人）的体验，选取来一个典型的用户和一个典型的场景作为呈现，帮助我们发现商业机会和挑战。这种地图的劣势是不能覆盖到全部用户并且主观性较强，优点是它能够非常聚焦某类用户进行深入调研。



4.偏重设计机会的体验地图

这类地图把定性、定量的调研过程详细地展现出来，最关注的是在过程中发现的新机遇以及有可能的体验提升。

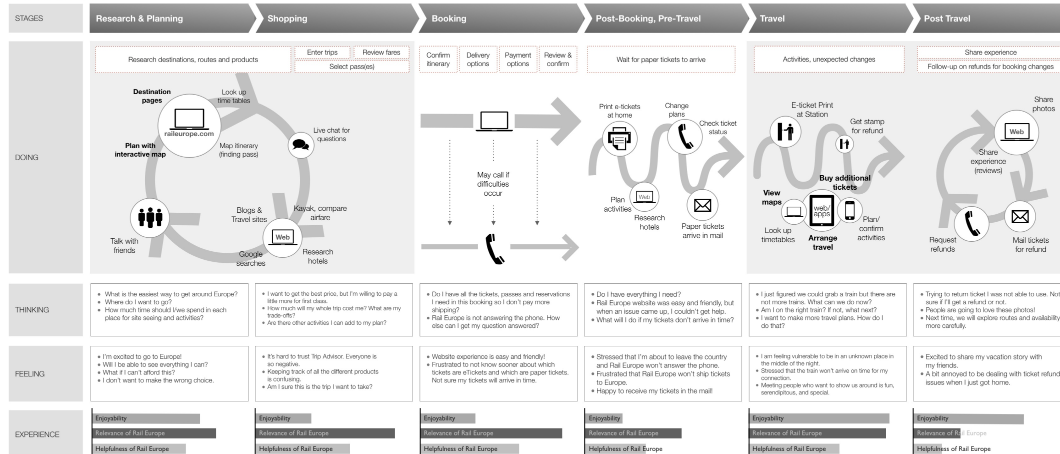
Rail Europe Experience Map

Guiding Principles

People choose rail travel because it is convenient, easy, and flexible.	Rail booking is only one part of people's larger travel process.	People build their travel plans over time.	People value service that is respectful, effective and personable.

Lens

Customer Journey

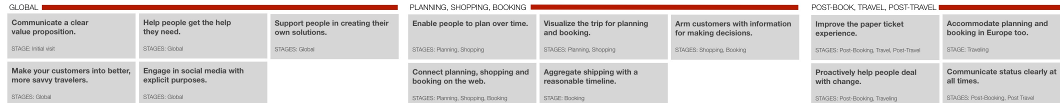


Journey Model

Qualitative Insights

Quantitative Informatioin

Opportunities



Takeaways

Information sources • Stakeholder interviews • Cognitive walkthroughs • Customer Experience Survey • Existing Rail Europe Documentation

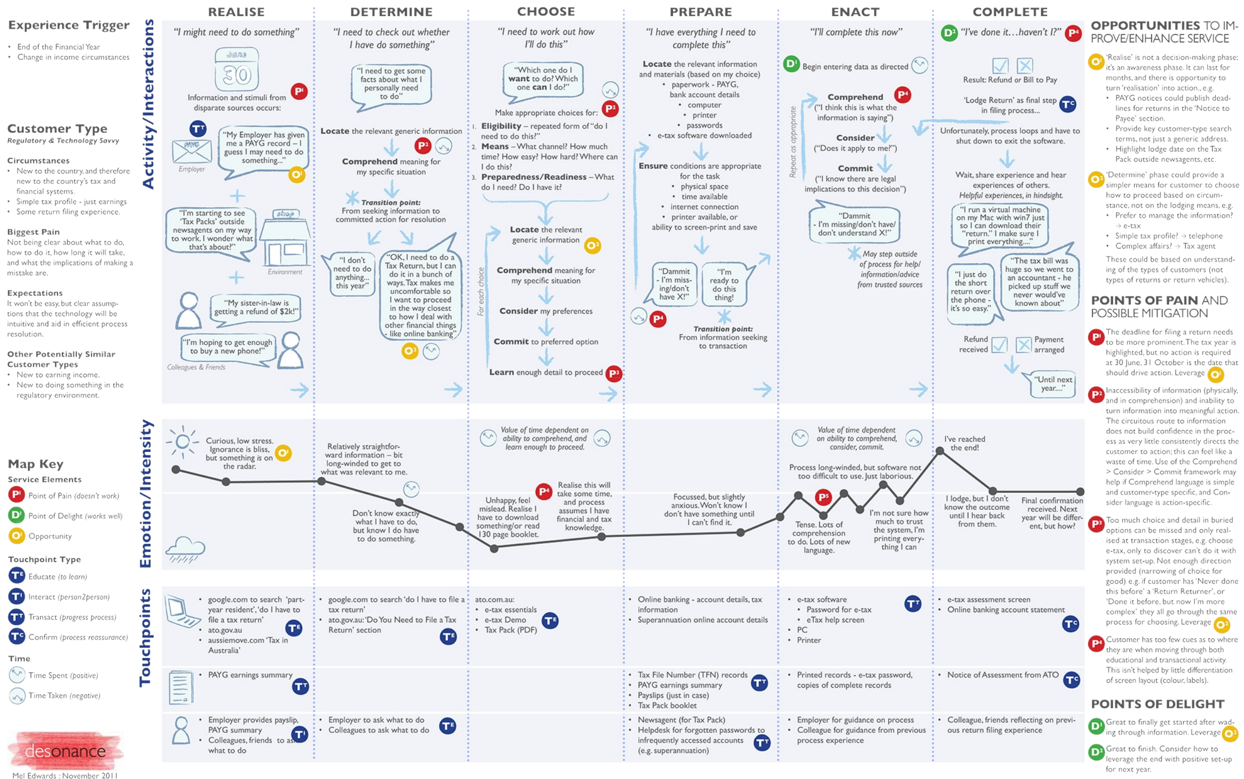
Ongoing, non-linear — Linear process — Non-linear, but time based

adaptive path

Experience Map for Rail Europe | August 2011

5.综合体验地图

这种选取了四类用户综合详细描述了他们的活动行为和接触点，在一张图上完整展现接触点的类型、痛点类型和机会类型。



工具网站推荐

最后，给大家推荐一些能快速输出用户体验地图的工具网站：

- https://www.smaply.com/
- https://uxpressia.com/
- https://maps.touchpointdashboard.com

一个查看各类map的网站：

- https://www.blankcanvas.io/

商业画布

“If you spend all your time looking at your competition,your product will look like your comprtition's ass”

设计

创意的产生和收敛

头脑风暴被视为一种激发创意的方案被广泛运用。在公司里，我们可能会不断地接触到各种各样的头脑风暴：可能是自己所研发的产品需要新的功能，可能是其他产品组的同事需要借你的“大脑”来想想新方案、可能是你需要为客户组织一个头脑风暴的workshop、可能是圣诞晚会需要想节目创意等等。总之，头脑风暴可能是最信手拈来、使用最频繁的创意手法之一。

然而，如果我们仔细一下，那些头脑风暴得出来的结论以及最后运用实际场景中的方案，就会发现，头脑风暴的效果甚微。以前常常经历的头脑风暴是这样的：上午突然接到通知说下午需要参加一个关于新产品的头脑风暴，在一个昏昏沉沉的午睡过后，来到一个会议室，4-6个人围坐在一个圆桌旁，被简单告知我们需要想下一个产品迭代的功能创意，每个人需要用10分钟写出5-10个创意，然后再用5-10分钟把创意用简单图形表达出来，接着是将方案show给大家看，让各自提出一些改进的意见，最后投票选出最好的方案。接下来的工作全部交给workshop组织者，他会把最好的方案做成ppt呈现给boss，让boss决定是否采用。

这个过程中，那些一个个便利贴上的“创意”就像是风中摇曳的蜡烛一般，面临着一次次被熄灭的考验：

- 在投票的过程中，创意需要迎合参与者的喜好，因为并没有制定什么参考的标准
- 在将创意制作成方案的过程中，组织者会将创意根据自己的想法再修改一次，也许他觉得最终选出来的创意不可能被通过，于是自己再想了一些迎合boss口味的方案
- 票选出来的创意可能常常是最“让人兴奋”的，但是这些创意常常没有考虑实际的限制，比如资金、资源、团队、时间等等，任何一个限制条件都能把这个方案打倒
- boss所参考的决策纬度可能与团队不一致，所以他可能会要求更多的方案，或者提出一个全新的方案，然后问：为什么你们没有想到这个？
- 等到实际的研发过程中，开发同事突然提出一些顾虑：这个创意太超前了，在时间范围内不太可能做得出来，或者即使做出来也比较粗糙。

所以，我们需要再重新想新的创意再一次次过兵斩将地向上汇报，还是硬着头皮加班加点地把方案做出来呢？

以上的案例足以说明：如果头脑风暴只是毫无章法地收集创意的话，最后会变成耽误产品进程、降低工作效率的方法。我们可能意识不到问题所在，陶醉在自己伟大的创意中，并且认为设计师就需要这样羡煞旁人的blingbiling的工作方式，将满墙彩色的便利贴拍成照片发到社交网络，以展现设计师的逼格所在。

其实，设计师最大的障碍不是别的，而是“理想主义”。因为设计毕竟不是简单的个人行为，只要是服务于目标群体，只要是团队协作，只要是商业目的的产品，都不应该体现个人的理想主义，否则会陷入不切实际的漩涡。设计方法应该是“逻辑的”“系统性的”“科学的”。

所以，头脑风暴到底应该如何做呢？

我们先看看头脑风暴可以产生的作用：

- 让团队对将来要做的产品或功能有深入的思考和了解
- 锻炼团队的思考问题的能力，跳出以往的经验和固有的思维模式
- 让决策者与团队有统一的认识
- 从头脑风暴中得出有切实有效的种子方案

有时【得出创意】如果只是目的之一，或者说，最终的创意只是所有科学过程的一个自然而然的结果的话，我们就更需要“科学地”看待这个方法。

一个比较体系的头脑风暴：

1.做充足的准备

- 知识储备：参与头脑风暴的人需要对所要解决的问题的知识领域有所了解，例如一个关于提高游泳馆淡季场地利用率的工作坊，应该去了解现有游泳馆的基本情况，如数量、规模、运营模式、人流量、经营状况等等，如果有条件的话，能够采访一些场馆经营者。创意说白了就是经验和感知，这些都将成为创意来源的材料。
- 流程准备：工作坊流程应该被事先设计好，可以以时间为依据，例如只有一个小时的时间，可以把时间分成背景介绍、过程、评估方案等若干模块，时间的限制可以帮助你决定哪些复杂的环节可以省掉或者简化；可以以目的为依据，例如工作坊的目的是产出大量的创意，那么创意的环节就可以占用2/3以上的时间，构想多种激发创意的方式。
- 设备准备：一些卡片、若干便利贴、一个拥有轻松氛围的会议室、需要把会议室的桌子椅子分成若干组，几面白墙或者白板、马克笔、大白纸、一张写有头脑风暴规则的海报等等。

2.邀请合适的人

合适的人指的是拥有专业背景的人、项目相关性高的人、少量决策者和少量专家、邀请的人尽量覆盖一个项目的各种角色。为什么是少量决策者和专家，因为这些人可能会影响参与者让他们的想法有所保留，觉得自己的想法不够权威而不敢发言，但是决策者和专家又是不可少的人，因为他们能够给出一些限制条件、他们更清楚项目的背景和领域知识。

3.一个井然有序的workshop

- 介绍背景：这些环节帮助我们了解所有的上下文，对工作坊的产出至关重要，绝对不可以简单对待。澄清我们所要解决的问题是什么，最好能给出一个具体的例子，让参与者了解他们所想出的解决方案能够对问题的解决带来哪些好处。 把问题的限制条件、优势资源和目标列出来，比如项目资金有多少，时间范围、技术限制，可以利用的资源有哪些，所要达到的目标是增量百分之多少或者是减少百分之多少的浪费。目标负责人是谁，这个角色在公司所处的位置、他所关注的事情、他的期待。
- 组织过程：介绍流程、时间--工作坊有几个环节，每个环节做什么，各需花费多少时间；在白板上写下一些规则，或者事先准备好写有规则的海报，如：一张便签纸只写一个方案、不事先对他人的方案进行批判、在时间范围内尽量多地写出方案、数量优先、写方案时保持独立不相互讨论、发言环节一次只称述一个方案...
- 分组： 把参与的人分成几组，每一组3-5个人，为什么是这个人数呢？因为超过5人的小组则会有人保持安静，少于3个人的小组则可能会较突出某个人的观点，无法形成客观的讨论。
- 痛点发散：让参与者根据之前的调研结果或者经验在便签纸上写出用户的痛点，例如：游泳场馆经营者想要优化场馆的运营模式，他们的痛点在于：游泳池安全隐患大、教练的流失率高、政府监管不到位、场馆淡季利用低、游泳池水质与成本无法协调等。每个人根据自己所了解的情况或者所处的位置写出痛点，注意提醒参与者不要写想象的痛点，这些痛点会分散注意力，把核心问题弱化。
- 痛点分类与投票：主持人将参与者所写的痛点分成几个大类，向所有人展示，统一大家的看法。接着让参与者投票，决定哪些痛点是接下来最需要被解决的。在投票之前，应该先把项目的限制条件和优势资源再重复一遍，让大家知道需要根据哪些标准来判断投票给哪些问题。
- 细化痛点产生的原因：这一环节至关重要，因为有一些问题很有可能超出了我们能解决的范畴，比如说政府监管不到位这类问题，它是一个非常痛的痛点，但是我们又对他无能为力，分析痛点背后的原因帮助我们了解它是否是一个“可以被解决的问题”。
- 写出每个痛点的解决方案：解决方案可以根据产生痛点的原因而来，一个问题产生的原因可能会有很多，有一些原因很重要，有一些则不那么重要，在提出解决方案的时候应该从重要、可解决的原因出发。
- 给解决方案分类、排序、投票：我们已经得到了大量的解决方案，我们可以将解决方案根据物理的、数字的、人的服务等纬度来分类，这个分类可以有效地帮助我们从不同的角度来解决问题，因为在很多老板眼里，数字化可以解决一切问题，动不动就要做一个网站或者app，但是事实上数字化只是众多解决方案中的一种方式，有些问题我们用人来解决可能是更好。例如机场的流程优化，不可能让所有的服务都数字化，墙上、地上、都在提示乘客下一步应该去哪里办什么手续，合理的方式应该是先用最简单的物理指示牌引导乘客，然后用数字屏幕来帮助他们找到对应的位置，最后如果还有疑问，则可以到咨询太询问服务员。这样就是物理的服务、数字的服务、人的服务三种方式结合起来，保证能效最大化、减少浪费。

- 评估方案：向大家介绍评估方案的纬度，根据这些纬度来给每个解决方案打分，例如：限制条件、资源优势、新颖度、解决痛点的程度、决策者的倾向、期望
- 优化方案：得分最高的方案不可能是十全十美的，我们需要列出这个方案的劣势在哪里，如果实施需要哪些条件，想出一些方案帮助我们把创意变得更好。
- 结束workshop：结束工作坊时，留下一些问题给大家思考，这样的好处在于能让参与者对问题进行深入思考，有可能会有更好的方案在会后被挖掘出来，如果有下一次相关问题的工作坊，这次的遗留问题是很好地参考资料。对工作坊进行总结，主持人陈述工作坊的过程以及最后的结论，确保每个人都能很好地了解到最后选出来的方案以及背后的参考标准。对工作坊本身的优化，让大家提出一些头脑风暴工作坊的改进意见，以便下一次更有效率。

4.给团队充足的反馈

创意的方案有可能被决策者通过或反驳，无论是什么结果，都应该第一时间反馈给参与者。被通过的，对参与者是一种莫大的激励；没有被通过的方案，参与者也可以从中知道，哪些因素是之前没有考虑到的，在下次工作坊当中可以加入进去。

写在最后

so，一个看起来有系统、考虑周全的头脑风暴就是这样被组织起来的，为什么说是“看起来”呢？因为每一个有效的工作坊都是需要不断打磨去适应实际的场景、人和环境的，每个人都应该需要根据基本原则去设计属于自己的头脑风暴工作坊。

信息架构

我们盖一栋大楼，需要花几个月的时间来画图纸、测量、计算，一切显得非常严谨。其实做一个互联网产品一点也不必盖一栋大楼简单，但是通常实际情况是一个公司的老板让设计师、开发、产品经理三个月内做出一个产品，而不经过仔细地构建，这就像直接让工人盖楼是一个道理。也许他们有能力直接造出一栋平房，但是对于建高楼大厦，则是一件不可能完成的任务。

今天我们所说的信息架构，就是数字信息界的建筑图纸，它用来帮助产品来搭建最初的结构，这个结构能够适应将来填充进来的内容，使得用户在使用产品时能够快速定位、寻找、搜索等。

如何定义信息架构：

- 产品信息内容的结构化设计、有效编排
- 是标签、导航、搜索的结合
- 它既是艺术的也是科学的，它设计内容信息和体验，让产品具有可用性和可搜索性
- 是从设计领域和建筑学领域的实践中借鉴过来的规范

通俗一点的说法就是，信息架构是帮助用户明白他在哪，他要找什么，周围有什么以及他决定要做什么的设计。

它包括几方面的内容

- 分类和构建信息
- 标签系统：如何呈现信息
- 导航系统：如何通过信息来浏览或者操作网站
- 搜索系统：如何寻找信息

想要做一个好的信息架构设计，我们必须先得弄清楚以下三者的内容和关系：

- 产品所处的环境：商业目标、资金、政策、文化、技术、资源、限制等
- 产品所包括的内容：数据类型、产品体量大小、现有的产品架构等
- 使用产品的用户：用户是谁、他们的目标和需求、用户习惯等

了解产品的五个层级：

表现层-框架层-结构层-范围层-战略层 （图）

其中，这五个层级都与信息架构相关，其中结构层和框架层是信息架构的体现，表现层是视觉设计的体现，范围层是数据内容的体现，战略层是产品目标和商业目标的体现。这五个层级，从下往上产生影响：商业策略决定了产品所包含的内容范围，产品的内容范围又决定了它使用什么层级机构和框架，框架对最终的视觉表现产生影响。

1.战略层：

也就是商业策略，始终离不开钱--不是帮公司赚钱，就是替公司省钱，我们可以参考"商业画布"那一章来制定。

2.范围层：

这里包含两种范围，一是工作的范围，一个是产品内容的范围。工作的范围是指现有的团队能够做哪些事情，哪些事情需要留到以后再做，包括沟通、合作、研发等等；产品内容范围是指产品上线需要完成的全部工作，包括数据、内容文档、子系统等等。完成范围层的设计的标准是，你必须有一份内容清单，并且对清单上的内容有优先级的排列。

3.结构层：

这一层主要是对所确定的内容进行管理、分类、排序，建立一个分类系统，它主要有两种模式，一种是由上而下的分类方法，一种是由下而上的分类方法。

由上而下：

我们按照已经明确了的产品的商业策略，将产品的内容归类，然后依次往下分出层级。比如我们要做一个在线书城，靠售卖电子书来盈利。那么我们就可以把在线书城这个商业目标分解成若干个小目标：艺术类书城、科技类书城、教育类书城，再分别将这些类别按照价格区间分类，如下图：

这样所产生的一个问题是，按照商业策略所分的类别也许和实际所拥有的资源不匹配，或者与用户期待不匹配。例如在艺术类书城下面，高价的书很多，低价的书很少，大多数目录都是空的，用户会对这样的网站产生内容不充实的感觉。

由下而上：

我们按照已有的资料来分类。假如现在有中国、日本、韩国、英国、芬兰、瑞典、美国、加拿大的书各有一万本，最直观的分类就是按照国家来分，然后再把中国、日本、韩国归为亚洲书城，把英国、芬兰、瑞典归为欧洲书城，把美国、加拿大归为美洲书城。如下图：

这样产生的问题是，分类并不能满足商业策略，因为商业数据显示，以书的类型来分类是用户最喜欢的方式，能带来良好的体验从而提高用户活跃度，按照国家的分类使这种优势消失。

两种方式有各自的优缺点，我们再实际的操作过程中需要不断地碰撞，使得商业目标和内容之间有一个很好地平衡，而不是非黑即白地一味满足商业或者一味从现有资源出发。例如我们发现这两种分类方式都不是很合适，可以采取第三种设计方式：先制定一个从商业上和从

内容资源上都接受的方案，即这个树状结构的中间点，设定按照书籍发售的时间，往上推把时间相同的艺术类书籍归在一起，向下推把相同时间不同国家的书籍分别归类。

把基本的分类做好之后，就要考虑这些分类之间的关系，它们也许是父子关系，也许是同级关系

- 树状结构：是最常见的一种结构，是不断收拢或者不断发散的结构。很多内容型的网站会用到这种模式，例如亚马逊、京东、网易新闻等等。（图）
- 矩阵结构：它不是一条胡同走到底的方式，而是在路径之间相互连接，这样可以满足不同用户的不同需求。例如有的用户喜欢按照类型查看商品，有的用户喜欢按照时间查看商品，单招树状结构，类型和时间的分类一定会有先后顺序--先按类型分，再按时间分。但矩阵结构可以在第一层级同时提供这两种方式让用户进行切换。（图）
- 自然结构：这种结构给用户一种全新的探索方式，它分类与分类之间待着某种惊喜和随机性。例如一些非常有个性的设计师个人网站，一开始出现动画，动画上有导航，从导航进去以后可能是他写的一段话，再点这段话到他的作品集，点击作品集又出现另一些动画等等。这种方式非常自由，它能给用户营造探索感，对于内容比较简单的产品比较适用。如果产品本身分类很复杂，又用了自由结构的方式，那对用户来说就是灾难。（图）
- 线性结构：一条线，没有分支的结构。比较适合看一篇文章、读一本书的产品。（图）

不管你用的哪种结构，最好都把它画出来，然后为每个模块去一个名字：（图）

4.框架层

如果说结构层是偏向于立体的结构，它看上去像是一栋大楼的钢筋结构，那么，框架层就是这个大楼每一层的设计。既然我们已经在结构层确定了每一层楼放什么，那么在框架层要做的事情就是讲这些物品按照一定的方式摆放，它包括三方面的设计：界面设计、导航设计、信息设计。

- 界面设计 界面设计要解决的最大的问题就是，当用户看到这个界面时，能不能马上找到他想要的东西。或者是，当用户无所事事闲逛的时候，能不能让某些内容进入他的视线和引起他的注意。我们最常用的一种方式就是把主要信息放在最显眼的位置并且放最大，如支付宝的界面：把扫一扫、付款、卡劵和咻一咻放到了最显眼的位置。值得注意的是，如果在导航上面放四个内容，其实不能让用户很好地聚焦，那为什么支付宝要放如此多呢？我们可以知道扫一扫和付款一定是用户频率最高的功能，另外的卡劵和咻一咻使用频率很低但由于是战略型的布局，所以不得不也把它放在了最显眼的位置。可见，界面设计不仅是要从用户角度考虑，更多时候商业的角度也是非常重要的一个方面。
- 导航设计 导航设计也就像是一个网站的地图设计，你需要通过导航来告诉用户：“你现在在哪，你能去哪”。导航设计是呈现给用户看的产品内容结构、分类图，也是帮助用户明确页面内容、了解所处位置的标注图。 导航大体上可以分为：全局导航、局部导航、辅

助导航、上下文导航、友好导航、网站地图导航等。

全局导航：你可以在任何一个位置去到你想去的任何地方 局部导航：当你进入到某个区域时，你只能在固定区域内任意穿梭，要想去到其他的区域，得先返回到主页面。 辅助导航：在一个位置上，提供另一个位置的快速入口，比如在某个界面都去到另一个深层级的页面，例如一个网上商城你进入到服饰的导航页面挑选衣服，此时页面上除了展示服饰产品外，还提供有鞋包分类下运动鞋的链接，此时就可以直接跳转过去。 上下文导航：在一个位置上，提供相关信息的链接，例如当你看完一个新闻时，底部会有相关新闻让你跳转。 友好导航：在页面上提供用户他们可能平时并不需要的链接，例如，每个页面都会有产品母公司的联系方式。

- 信息设计 界面上除了导航和控件元素之外，很重要的东西就是信息的设计，特别是对于信息类的网站，这类信息显得尤为重要。对于一个产品，你要展现哪些信息，哪些信息对用户最有效，它们之间的关系应该如何呈现，这些都是信息设计所要考虑的事情。如图：

5.表现层：

就是我们常说的视觉设计，主要是对用户的感知进行设计，让用户愉悦、让企业的品牌得到传播。具体的设计方法可以参照视觉设计那一章。

视觉设计

视觉设计常常被认为是非常主观的设计，因为不同的人审美差异非常大。就像听音乐，农民工听的音乐和大学教授听的音乐几乎不太可能一样，他们所生活的环境、接触的事物和思维方式决定了他们的音乐审美。然而世界上有高雅音乐，也有低俗音乐，只要有人喜欢它就有存在的理由。做视觉设计也是一样，我们不能单纯地以美和丑来评价一个产品视觉设计的好坏，而是应该以它所服务的人群是否接受并喜欢此风格来评价，更进一步说，作为一个有责任心的设计，设计出来的作品应该把观众的审美向更高的层次培养。

今天我所要谈论的视觉设计，它特指的是互联网产品的视觉设计，所以我们首先要注意到它的“互联网性质”，其次是它的开始、过程以及结果。互联网中的视觉设计，我认为值得强调的两点是：

1.它是一种感知的设计

2.它是产品设计中的一环，不可独立于产品之外。

接下来，我会一一为大家解释这两个观点。

视觉设计是一种感知和服务的设计

在做任何具体的设计之前，最重要的事情就是要知道目标是什么。那视觉设计的目标是什么？如果你的回答是：“美”，那说明你是一个对美有追求，并且停留在初级阶段的设计师。我认为，视觉设计的目标有这么几个：

- 满足基本的设计原则
- 让你的老板和你的用户同时喜欢它
- 在有限时间内，做到效率最高，投入产出比最大

第一点就不用说了，如果产品不是做行业内大的设计创新，就没有必要颠覆现有的使用了多年的设计原则。那么，符合基本的设计原则是设计师的底线。这个原则包括：每种尺寸的屏幕使用合适的字重、用色和图形与产品的调性一致、考虑用户使用的场景、整体风格与竞争对手的差异性等等。

第二点和第三点，其实都是在做一种感知的设计。

我所看到很多设计师会这样做视觉设计：

了解产品目标**-**确定产品风格**-**寻找与风格匹配的视觉参考资料**-**视觉稿输出**-**修改**1-**修改**2-**修改**N-**截稿上线。

这样做貌似也没有什么问题，但是，我敢肯定，有70%的时间是花在修改上的，这个修改来自于客户或者老板的建议，他们常常会以“不够大气”“没有感觉”“有没有更多的方案”来让设计师重做设计。之所以会产生这些工作量，究其原因，是因为设计师和老板之间出现了审美

差，大家心中各自有一个骄傲的“审美观”，却相互看不上对方。

这时候，如果我们继续不断地出新方案或修改之前的方案，很有可能因此而浪费时间，到了上线的前一秒还在忙前忙后，效率十分低下，投入产出比小。

如果我们换一种方式来做思考呢？如上所说，视觉设计需要我们把所有人的审美统一一起来，即使不能真正的统一，也可以做到大概的一致性。这些需要我们在做视觉设计的每一步都考虑到“感觉”的设计。怎么做呢？

- 1.首先，为了缩小大家的“审美差”，我们可以做一场关于视觉设计的工作坊，先了解客户或老板的产品视觉倾向是什么、期待是什么。
- 2.接着，与老板或客户共同建立一种设计评判的标准，目的是让大家用同一套评价系统来评价设计，把“感知”控制在一定范围内。
- 3.当“审美差”缩小，“设计规范”建立好之后，就可以在一个比较确定的方向上做2-3套方案。这2-3套方案的方向可以有很多组合的方式，可以按照具体项目的不同来设定，例如：如果老板【期待】有创新，组合的方式可以是：2套方向之内的，1套方向之外的。
- 4.做设计展示，让所有利益相关者参与，展示方案时很关键的一点是把规范和之前的工作坊结果再展示一遍，再次把所有人的“感知”控制在范围之内。

我们发现，需要花精力的地方有时并不仅仅在于视觉方案本身，而是在于这些“感官”设计来与客户达成统一、增强客户的参与感、展示设计师的专业度。这样的设计，可以大大缩短我们修改的时间。

视觉设计是产品设计中的一环，不可独立于产品之外

一些设计师貌似对美有很高的追求，以至于会花费大量时间在视觉设计的细节打磨上。并不是说这些做法不好，而是在那之前，我们要了解到，视觉设计是产品设计中的一环，它应该跟随产品策略去做，而不是单独存在。 那么，什么是产品策略呢？每个产品的策略不同，但是我们大概可以知道它们都会包含哪些内容:

- 用户群
- 产品方向
- 竞争对手
- 时间规划
- ...

这些看似与视觉设计没有什么关系的，却是最值得视觉设计师重视的。

- 用户群决定了我们为谁做设计，而不是为自己做设计
- 产品方向决定了视觉风格应该呈现什么感觉，比如美食类产品的设计就应该多用暖色+图片的形式，容易勾起用户的食欲。
- 竞争对手决定了我们是否应该跟他做差异化设计，或者一致性设计。有些竞争对手在市场上做得非常成功，这时候，如果做差异化小的设计，或许可以让用户产生好感，在商业上是成功的策略。

- 时间规划决定了我们能用多少时间来做设计，这个时间应该包括：风格调研-设计-汇报-修改等步骤，设计师应该把所有的时间节点规划起来，而不是只规划打开ps的那段时间。

对于一个to C的产品而言，视觉设计是非常重要的，因为市面上大多数toC的产品界面都设计得非常美观，用户的审美已经被提高，分配更多的时间到视觉优化上，是值得做的事情。

对于一个to B的产品而言，视觉设计的优先级就没那么高了，因为to B 产品最最重视的是数据的准确性、产品使用的高效性、流程之间的通畅性等，留给视觉设计的时间有时会非常短，这时我们如果想要做一个精致的视觉是不太可能的，也许买一套"视觉模板"，再在上面做相应的修改，是投入产出比最高的。相信很多设计师都不能接受这种方式，但是，我们做任何设计都应考虑到全局，懂得在限制下做设计。

视觉工作坊

视觉工作坊是视觉设计最重要的一环，它是一个与利益相关者沟通的机会，也是一个统一大家意见的契机。这里展示一个视觉工作坊的流程，帮助大家了解如何组织一个视觉工作坊：

1. 规划工作坊的流程，确定邀请对象，确定时间 制作一个ppt来展示流程，让参与者能够清晰的知道每一步应该做什么； 确定邀请对象是一个至关重要的事情，必须邀请到最终能决定视觉设计的角色，否则进行了几个小时的工作坊就白费了； 时间最好控制在1-1.5个小时之内，如果时间太长，则会成为某些角色拒绝参加的理由。

2. 工作坊开始，首先介绍工作坊的目的。 很多参与者这时候都是第一次接触视觉工作坊，明确工作坊的目的、说明大概要花费的时间、包含哪几个设计活动、最终的产出物等，这些介绍能够给参与者安全感，让他们高效地完成工作坊的活动。

3. 对于现有产品的视觉评估 让每个参与者把脑海中对于产品现阶段的视觉设计做一个评估，将评估出来的关键词写在便签纸上。这个活动主要用来发现问题寻找痛点。大家一定是对现有的视觉设计不满，才会想要设计新的视觉，视觉化的工作坊也让大家可以看到问题集中在哪几个方面，那么在之后的视觉中就能集中解决这些问题。

4. 对于产品未来的视觉评估： 让每个参与者把脑海中对于产品将来的视觉设计做一个评估，将评估出来的关键词写在便签纸上。通过这个活动，我们可以知道参与者对于未来视觉方案的猜想以及他们的偏好，也能从关键词当中更加了解他们想把产品做成什么形态。参与者常常会将自己心目中认为好的产品的视觉关键词写下来，这些信息对做方案非常有利。

5. 让参与者选出两张他们认为与未来风格匹配的图片，并解释原因。 我们知道，从文字的描述到图像的描述，是有一个鸿沟的，比如按照"活泼"这个关键词来设计视觉风格，可以设计出成千上万种活泼的视觉方案，所以我们要知道的是参与者心中对于"活泼"的想象是什么，选出与关键词对应的图片能够进一步收敛他们的想象，并且实现了第一次视觉的"具象化"。

6. 从事先准备的界面效果图当中，选出与图片最匹配的一组。 组织者需要事先挑选出3-4套视觉风格并归类：如活力、专业、极简等，贴在墙上，并且贴上是什么元素让这些风格传达得如此到位，比如活力的风格，常常用彩色、圆角、灵活的排版形式来表达。从图像到界面其实是第二道鸿沟，参与者常常不具有设计背景，所以虽然知道自己喜欢什么风格，但很难想象怎么去表现它，用投票选择的方式去选择一种风格，让他们了解用什么元素去表达这种风格是最合适的，这些元素可能也会在将来的设计中运用到。这实现了第二次视觉的“具象化”。

通过这六个步骤，我们引导大家从【未来视觉设计的想象】具象成【某一种风格的视觉页面】，这样做一方面能够让全部参与者达成视觉的一致，另一方面让他们了解到视觉设计是一个从模糊到具象的过程，这个过程既鼓励大家发挥想象，又用种种方式收敛成可落地的方案。

写在最后

无论是什么设计，都要求设计师有“出跳”的能力，视觉设计也是一样，首先要跳出【视觉设计】看问题，看到视觉设计之外的东西，如时间、流程、目标等。下一步，需要看到设计与设计之间的不同，这类设计和那类设计，他们应该用什么样的方式呈现最好，是不是最好看的永远是最好的。最后是知道如何达到设计目标，如工作坊，都是不错的值得参考的方式。

测试

思维转变

传统意义上的开发者，往往和 `geek` 关联在一起。即热衷技术的怪人，他们不修边幅，沉迷技术，难以和正常人打交道。现实世界中，这种开发者事实上是比较少见的。以我自己的经验，反而是那些总觉的技术是改变世界，拯救世界的唯一方式的人多一些。

他们往往自视甚高，不大看得上那些产品经理，项目负责人，测试经理等带有 `管理` 头衔的其他同事，当然也不太看得上其他的同行。更别提那些IT系统做的非常差，但是有莫名其妙的非常有钱的企业，他们简直不配和这些程序员共享同一片蓝天。

他们会嘲笑这些企业使用的过时的源码控制系统，也会嘲笑他们把主机锁起来免得被员工偷走，但是对这些企业的精巧，高效的商业模式视而不见。

成为咨询师

本文旨在帮助`开发`完成向`咨询师`的转变，内容不但涉及向`UX`学习，还包括思维方式的转变。我尽量采用一些亲历的例子来说明该如何做，也会适当的解释为什么需要这样做。不过在展开详细讨论之前，首先来澄清这里提到的三种角色。

开发（**Developer**）角色

`开发`是指那些喜欢写代码，享受写代码，喜欢纯粹，讨厌办公室政治，永远穿T恤的有些偏执的程序员。跟他们打交道，有这样一些注意事项：

- 不要让他们帮你盗`QQ`号
- 不要让他们帮你修电脑或者装Windows系统
- 不要跟他们讨论`人文/政治`类的问题

`开发`往往还单纯的可爱，除此之外，他们还有这样一些特点：

- 逻辑清晰
- 与人争辩时往往可以通过清晰的逻辑而获胜
- 单身

业界已经有很多关于`开发`的描述了，我这里也有一个描述`开发`的列表：

~~当然，要严格界定一个人是不是`开发`是非常困难的，大多数情况下，他们沉默寡言，遇到程序中的bug或者在调试某些库的问题时眼神呆滞，口中念念有词，他们不太喜欢和陌生人说话，在晚上精神充沛，白天则显得有些呆滞，喜欢喝咖啡，相信世界上有绝对的正确和错误，往往会带着非黑即白的二分法来看待事物，生活很难自理，喜欢机械键盘/电子设备，周末宁愿宅在家里写代码也不去做社交……~~

用户体验设计师（**UX**）

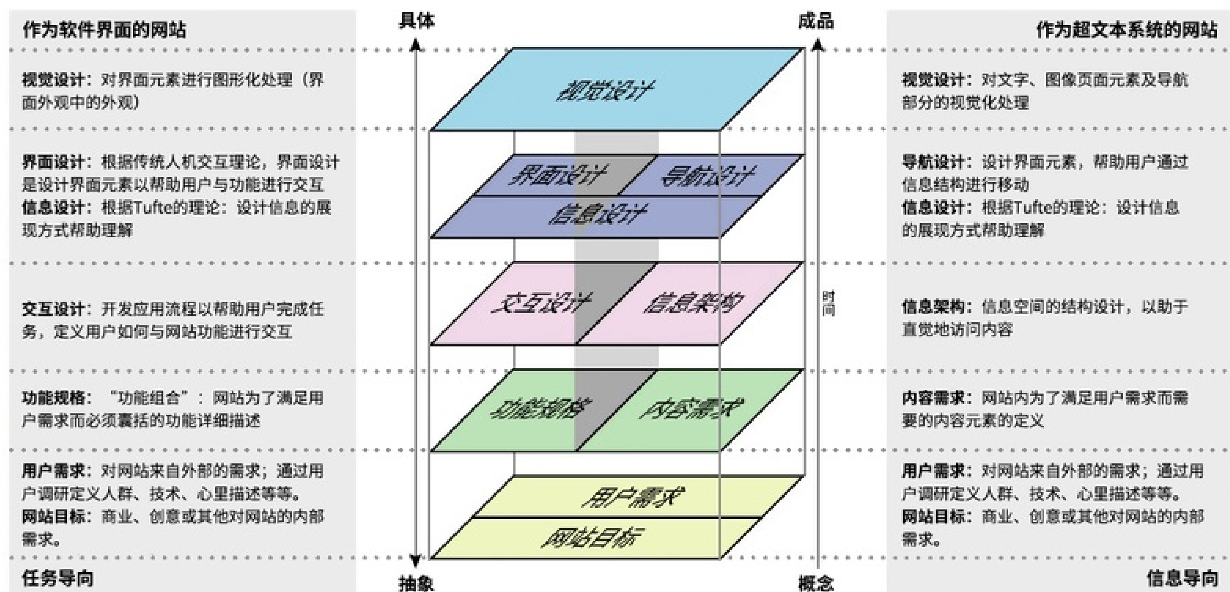
`UX`是指用户体验设计师，在本文的上下文中，更偏向与非`视觉设计`的那些设计师（产品设计师）。在项目中，他们会做用户调研，竞品分析，信息架构简历，交互设计（纸上原型，低保真）等活动，并负责开发纸上原型，验证这些原型等。

和`UX`打交道，也有一些应该注意的点，比如：

- 不要叫他们美工
- 不要对他们说诸如：“帮我美化一下这个页面”，“这个颜色得再亮一些”之类的话
- 不要跟他们讲关于程序员的笑话

事实上，人们对 UX 的误解很深。提到 UX 人们的第一反应是 PhotoShop ，P图/切图。这仅仅是他们日常工作中很小的一部分。大部分 UX 还要做很多用户研究，信息架构整理的事情。老实说，我在去年5月之前的对 UX 的认识和大部分 开发 的认识是一样的，但是在后来的项目上和多个 UX 合作过之后，我彻底改变了原先那种偏见，开始敬佩他们，并向他们学习。

设计工作可以细分为这样一些不同的方面（图片来源网络）：



UX 的一项特别的技能在于能从复杂的现实世界中抽象出清晰的信息（用户画像，体验地图甚至最后的用户故事）。这项技能不但重要，而且还很牛逼。

知识的诅咒

《反脆弱》里有个有意思的例子：人们仅仅创造了非常有限的词汇来描述颜色，比如蓝色，红色，而任何一个视觉正常的人都可以轻松的识别出数百种不同的颜色。也就是说，人们可以很轻松的理解相当复杂的事物，但是很难向别人描述该事物（想象一下向别人描述一只 章鱼 的颜色）。

人们对于现实世界中的事情（特别是复杂的业务场景）往往只能意会而很难言传，再加上知识的诅咒（我在《如何写一本书》里，详细讨论了这种常见的陷阱）的存在，当用户在描述A的时候，在没有上下文的人听来，很可能是B或者C。这种情况在软件开发中非常常见，也是很多项目之所以延期的原因（大量并无必要的返工，需求澄清等）。

在项目前期，`UX`需要和客户坐在一起，将客户的需求分析清晰。分析细节包括业务场景，用户画像生成，信息架构，体验地图等等，这些信息并不是天然就显现的，恰恰相反，它们需要UX经过很多轮的辛苦引导，从用户的脑海里`提取`出来的。

这里需要`UX`的核心能力是：

- 有目的的抛出问题，引导客户进行发散
- 有节奏的收敛，形成共识
- 不断修正过程中的错误
- 可视化能力（这可能是大部分人觉得唯一和UX相关的点）

咨询师

`咨询师`是指那些根据自己的丰富经验来帮助客户解决具体问题的人。这些问题并不一定局限在技术上 —— 比如架构的设计，具体前端/后端技术的选定，还包括一些流程的改善。比如引入新的`工程实践`来缩减项目的周期时间，帮助团队发现问题，建设团队的能力，作为各个团队间的润滑剂帮助项目成功等等。

`咨询师`工作中的一个常见的场景是：

- 列出目前遇到的问题
- 确定各个问题的优先级（和各个利益方）
- 制定方案
- 给方案加上时间，形成计划
- 细化计划中的条目，并促成它

引导/启发

我在印度的某一期`TWU`当教练的时候，发现了一个很有意思的现象，国外的同事在组织培训时更强调用`引导`/`启发`的方式，让学生们自己得出结论，并在课堂上进行讨论，以期教学相长。只有在过程中有`启而不发`的情况出现时，教练才会适当抛出自己的开发，并再次启动讨论。



与我一直的认识不同的是，这种方式效果很好。通过一些适当的启发，学生很容易自己讨论出一些有趣的看法，然后教练在这个基础上做一些总结，并帮助他们分析不同看法/想法之间的优劣。

我非常认同这种模式，后来自己组织的其他培训/workshop也都尽量采取这种方式。咨询师在客户现场，也应该采取这种 `引导` 的方式帮助团队来完成能力建设，而不是事必躬亲。

角色转化

从 `开发者` 视角切换到 `咨询师` 的第一要诀就是：让团队解决自己遇到的问题！乍听起来，`咨询师` 好像变成一个多余的角色了：既然团队自己可以搞定，还要 `咨询师` 干什么呢？ `咨询师` 的职责是让团队意识到问题，理清思路，制定解决方案，并逐步实施。

使能/赋能

我们来看一个简单的例子：在客户现场，你发现团队往往在集成时会花费很多额外的时间和返工，开发过程中大家各自为政，没有人知道一次commit会给软件包造成什么影响。

如果你是一个 `咨询师` ，应该如何解决这个问题？一个常犯的错误是，直接上手帮助团队搭建持续集成（CI）环境，并设置CI纪律（比如build红了不许过夜，红的时候其他人都不许commit等）。

一种更好的做法是：做为 `咨询师` ，首先需要帮助团队认识到这个问题，你需要让所有人都知道，我们现在的问题是什么。在所有人都清楚了这一点之后，你需要提出（或者 `引导` 出）持续集成的概念（因为根据经验，这是一种可以很好的解决集成时额外的返工现象的好办法）。

但是对于不熟悉 `持续集成` 的团队来说，搭建一个持续集成环境是一个非常 `复杂` 的任务。因此你需要分解这个任务为一些更小的，可以被解决的问题。

- 申请虚拟机资源
- 安装jenkins（包括安装JVM，创建用户等）
- 配置本地构建脚本到jenkins（构建脚本，自动化测试等）
- 申请显示器资源（作为CI Monitor）
- 将结果显式在CI Monitor上

有了任务之后，你需要分别为这些子任务分配owner。对比搭建 `持续集成环境` 这样的大任务，这些小的任务已经非常具体，更重要的是，他可以被团队中任何人理解并解决。

学习做引导

除了思维方式的转变，以及自身过硬的专业技能（比如clean code/重构能力，自动化测试，DevOps，持续交付经验等）之外，开发者需要从 `UX` 那里学习如何发现问题，并将问题可视化出来的技能。

当你发现团队面临某个问题是，可以通过组织一个类似 `头脑风暴` 的会议来帮助团队梳理：

- 提出问题
- 维护会议纪律，保证所有人都贡献自己的想法
- 将想法/问题归类
- 找出问题的解决方案
- 制定计划（包括时间点和owner）

关于如何做引导的详细信息，还可以参考我的上一篇文章。

进一步的阅读

除了上边提到的

1. 思维方式的转变
2. 向 `UX` 学习引导的技巧

之外，事实上还有很多技巧和内容需要学习：

- 《引导的秘诀》
- 《视觉会议》
- 《第五项修炼》
- 《系统思考》

当我们谈论引导时，我们谈些什么？

什么是引导（facilitation）

引导（facilitation）的词根来源与拉丁语“facil”，意思是“让……更容易”。而负责引导的`引导师`（facilitator）的核心职责是，通过一系列的活动、技巧，保证引导会议顺畅的进行，并解决整个过程中的问题，使得参与者就问题产生一个共识，达成一个结论。

其中可能涉及很多具体的问题，比如几乎在每个会议中都可能看到的：

- 如果有人尝试将会议变成一言堂，如何处理？
- 如果参与者不愿意分享自己的观点，如何处理？
- 过程中，两个参与者产生了争执，如何处理？
- 如何把握节奏，刺激与会者发散？
- 如何在收集到足够信息后，进行收敛？

显然，这是一个技术活儿。一次好的引导可以将与会者的众多想法，信息聚合起来，形成对团队下一步要做什么有极强指导意义的`方案`。

日常的引导活动

在平时的工作中，我们其实已经在频繁的使用引导活动，但是很少有人将其作为体系来关注，也很少有人能将这个能力应用在其他方面（比如在客户现场咨询，或者参加售前等）。引导是如此的常见，以至于我们对其视而不见。比如在interview完成之后，所有面试官和HR一起做的well/less well的列举；各种社区活动（Open Party，CDConf等）之后的回顾；每个项目在一个迭代结束后的Retro；对于某个问题的头脑风暴等等。

项目回顾会议

在开始前，引导师需要保证团队：

- 每个人都有开发的态度
- 整个过程需要在一个足够安全的环境中进行（Safe Check）

有时候，有Team Lead在场，新人可能不愿意对某事（比如最近加班有点过分）发表自己的看法等。这时候需要有`Safe Check`，比如分为1到5档，大家用不记名投票的方式来表述自己是否觉得安全。如果投票结果显示大部分人都觉得不安全，则需要与会的人中，职位最高的那个人离开会议，然后再做一次`Safe Check`，直到大家都觉得足够安全。不过，对于已经进行过多轮回顾的团队，我们往往会忽略掉这一步。

Retro过程是，团队坐在一起，回顾上一个迭代（通常是两个星期）做过的事情，有哪些做的比较好，哪些有待改进，有哪些疑惑等等。Retro可以有很多的形式，比如简单的 `Well/Less Well/Questions` ，更聚焦在产生 `Action` 的海星式等等。



通常的顺序是：

1. 引导者请大家用纸笔将想法写在便签（stick）上
2. Time box这个过程（通常是5分钟）
3. 大家将这些stick贴在墙上
4. 引导者和团队一起过一遍所有的stick
5. 归类相似的stick
6. 引导者促进团队交流，讨论stick上的问题，并形成一些改进点（Action）

Action一定要足够具体，并且需要一个所有者， `所有者` 负责确保该 `Action` 一定会发生。比如团队发现上一个迭代中 `Code review` 做的不够好，一个 `Action` 就是每天下午5点有人来提醒大家来进行 `Code review` 。

如果这时候发现有太多的问题，团队可以用投票的方式选出本次Retro要讨论的数个stick。

引导会议

在日常工作中，我们几乎每天都有会议，而且越来越多的团队已经意识到冗长，无聊的会议有多大的杀伤力了。在很多会议上，与会者要么在刷新朋友圈，要么在对着笔记本电脑写代码或者读新闻，即使强制要求不许带电脑和手机的情况下，也无法限制参加者神游太虚。

根据《引导的秘诀》这本书里的定义，引导会议是

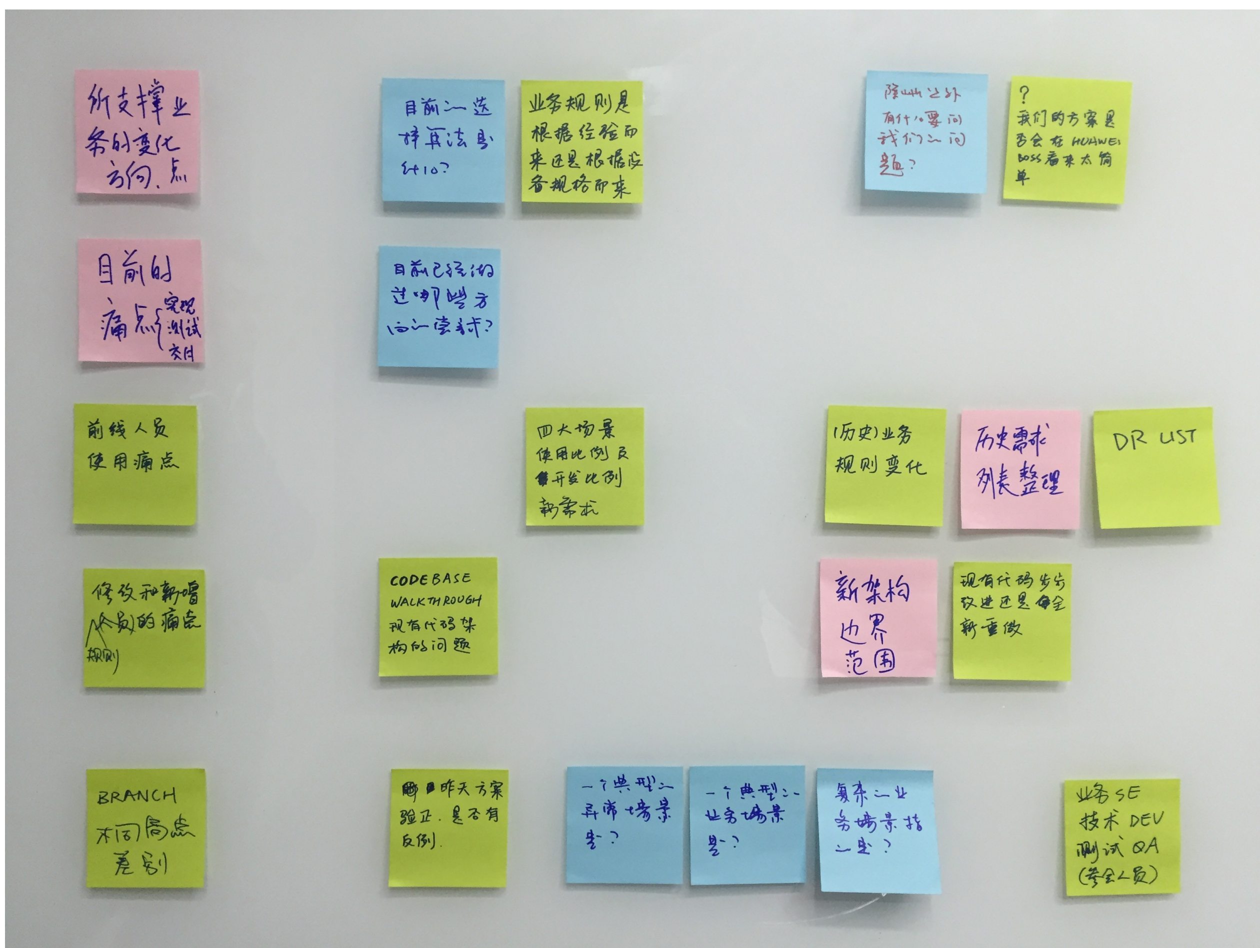
> 引导会议是一个高度结构化的会议，会议中的领导者（引导者）引导参会人通过预先设定好的步骤达成所有参会人产生，理解并接受的结果。

引导会议需要充分调用参与者的积极性，每个人都需要足够聚焦，这要求引导者可以有能力使得团队振奋（比如幽默的风格，或者具有挑战性的问题等）。另外，每个人的idea都需要被充分重视（一个细节就是不要随意篡改你听到的内容，这是没有经验的引导者常犯的错误之一）。一旦所有参与者都积极起来，引导者就可以稍微退后一些，将舞台交给团队。

而有时候，情景则相反，大家都不发言，也没有看到明显的发言的趋势，这时候需要一些方法来激励。如果是团队都比较茫然，引导者需要列出一些简单而容易理解的步骤，帮助团队按照预设的节奏来逐步前进。比如，在一开始的时候就将agenda板书在墙上，并通过头脑风暴的方式，鼓励参与者来将自己的idea可视化出来。

一个典型的误区是，引导会议的最后结论是本来就存在与引导者脑海中的想法。如果仅从结果来看，这种情况可能发生，但是只能说是碰巧而已。一个好的引导者需要帮助与会者自己产生，并得出一个可行的，被广泛认可的方案，而不是强加一个自己的给团队。

我们最为专业的引导活动是UX团队在客户现场的`inception`，`inception`由一系列相互关联，环环相扣的工作坊组成，这些工作坊基本上都需要采用很多引导的技巧，帮助客户团队将自己的问题描述清楚，并形成一个所有参与者都达成一直的可行方案。



如果你不知道如何开始一个引导会议，一个简单而通用的模式是：

1. 我们的现状是
2. 我们的目标是
3. 我们如何到达目标
4. 在行进中，如何度量

《引导的秘诀》里还提到了一种`5P`模式：目的(Purpose)，产出(Product)，与会人(Participant)，可能的问题(Probable issues)以及流程(Process)。

5P提示你在准备会议之前，需要尝试回答这几个问题

- 为什么要开这次会议？主要目的是什么？
- 会议后的产出是什么？
- 谁需要参与会议？
- 在会议中，我们可能遇到什么问题？
- 遇到这些问题是，我们如何解决？

引导中的常用技巧

在引导活动中，有一些基本的规则，可以保证引导会议的顺畅性，比如

- 引导师需要有足够的权威（可以打断那些长篇大论，保证过程的流畅）
- 如果人数太多，可以使用token（比如一个玩具考拉，或者一个澳式橄榄球，只有持有token的人可以说话）
- 保持one conversation（不要交头接耳）
- 每张stick上只写一条问题/想法

引导师必须有控制会话何时结束的能力，否则引导活动将会变成一发不可收拾的冗长会议。坚持`one conversation`可以保证参与者足够聚焦，也保证所有人都在同一个频道上。如果发现有交头接耳的，引导者可以直接打断并提醒之。

每张便签上只写一条想法，首先可以保证多样性，便于讨论，也便于后续的分类。另外，简洁的描述在一定程度上可以促进与会者进行讨论，而一个冗长的描述则会让人丧失兴趣。

另外还有一些比较基础的技巧：

- 所有讨论都应该对事不对人（特别是一些负面的总结）
- 如果有人提出与议题并不特别相关，但是又特别重要的点时，可以将这些点记下来（不要轻易打击发言者的积极性）
- 不定时的总结，以确保参与者都在同一频道，并且有助于大家对进度的了解（是不是快结束了）

积沙成塔

> 你在展望未来的时候，是不可能将这些片段完整的串起来的，你只能在回顾过往时才可以。所以你要相信，在未来某个时刻，你会将这些片段串起来！
>
> 史蒂夫.乔布斯

当我读完《程序员的思维修炼》时，有一种韦小宝和双儿终于八部《四十二章》中的地图串起来的感觉。那些原本琐碎，散落的片段开始汇集，并形成一条明晰的线。很多原来遇到的问题和困惑也都变得清明而鲜活。

不要理解错了，我可不是说这本书本身有那么大的功效。事实上，这种非常美妙的体验与之前的一些阅读有很大的关系，这段时间我在读/重读的一些书是：

- 《发布！软件的设计与部署》
- 《反脆弱》
- 《系统思考》
- 《第五项修炼》
- 《精益开发实战》
- 《持续交付》
- 《凤凰项目：一个IT运维的传奇故事》
- 《认知与设计》

不知道为什么，原标题为《Pragmatic Thinking and Learning》的书，为什么会被翻译为《程序员的思维修炼》，整本书也基本上没有什么程序员，软件开发相关的内容。两年前我读了几页就放下了，我以为又是一本《从小工到专家》，《Clean Coder》之类的书。

核心观念

事实上，《程序员的思维修炼》里在讨论从大脑以及人类思维的方式。它强调使用一些实践来提升大脑的工作效率，包括如何平衡使用R(Rich)型思维和L(Linear)型思维（不是简单的左右脑思维，人类的神经系统比简单的左右脑半球可要复杂的多了）。

其中为了锻炼R型（基于图像而非语言，整体而非局部，感觉而非理性）的思维，作者建议做一些编程、写作之外的练习。

比如我经常做的有：

- 练习某种武术（比如形意拳）
- 绘画（素描，练习手眼配合）
- 把想到的东西说出来
- 听相声（在听相声的过程中，你会不自觉的脑部画面）

这些看似与思维方式，具体的编码能力没有关系的事情，其实会在很大程度上帮助你成为更加优秀的`问题解决者`。

在实际工作中，我们会遇到很多问题。这些问题在最开始时我们可能只是全盘接受，而没有产生深层次的思考。

比如为什么要结对编程？持续集成/持续交付到底要做到什么程度？如果项目压力很大的话，还要坚持编写自动化测试吗？如果客户的组织结构限制，要裁剪掉一些实践，如何做平衡？

一个部门墙的例子

我去年在客户现场做一个项目的售前，在帮助客户梳理需求的时候，遇到一个在我看来十分奇怪的需求：客户的开发经理想要开发一个`接口管控`系统。

客户团队在开发过程中感觉很痛苦：正在开发的业务系统分为很多个组件，每个组件都由一个独立的团队开发，所有的组件会定期的做集成。在集成的时候（好几周一次），经常会有某些接口更改之后导致依赖方编译失败的情况。也就是说，接口的变更很难控制，依赖方基于旧版本的接口进行开发，到集成的时候发现两者的不匹配，由于没有人知道接口变更对其他人的影响，排查问题会花费很长时间。

而好几周一次的集成，往往以为着开发时间的结束，软件要进入测试部门的验收测试。测试部门要验收，又没有一个可用的版本，所有压力就会跑到开发部门，而这个排查过程又很费时，以为着有人需要加班工作，所以大家对接口变更深恶痛绝。

在这个背景下，客户经理想要一个`接口管控`系统来保证所有的接口定义（C语言的头文件，或者Java里的接口）都不能被自由的变更，这样集成的时候就不会有问题了。而如果是在一定要修改的时候，通过这个管控系统，开发经理可以知道谁，想要在何时，发起什么样的变更，而影响的依赖方有哪些等等信息。以后还可以对这些变更进行统计，分析哪些接口是多变而不稳定的。

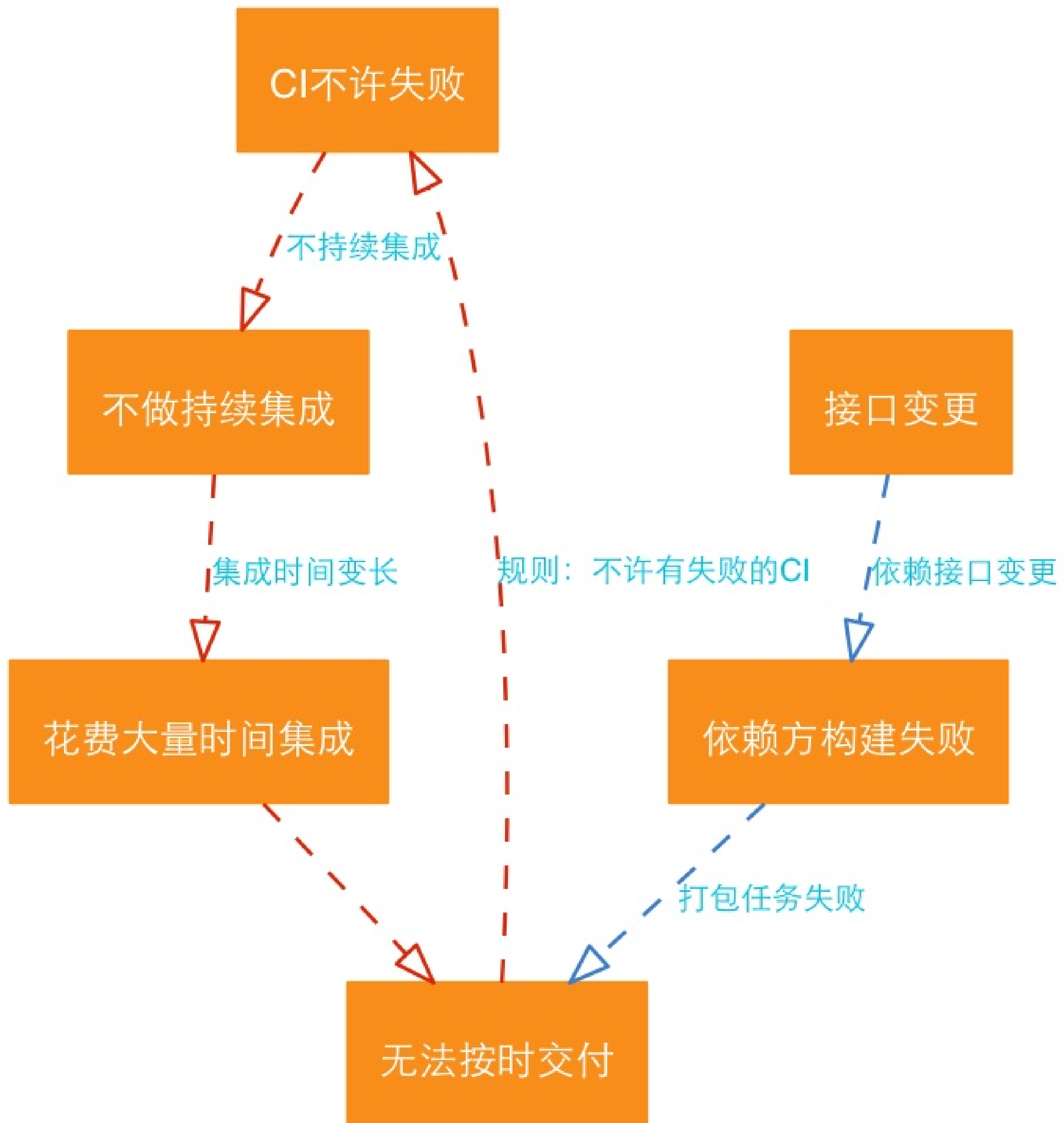
如果忽视这个需求背后的不合理的本质，开发者很可能会眞的建模出一个表示`接口`的实体，然后开发一个信息管理系统来管理这些实体，还可能为其开发`dashboard`用以展现接口的变更分析等。

我看了看客户团队搭建的jenkins的监视器，问他：你们不是有持续集成环境吗，为什么不把所有的组件的集成都放在CI上运行呢？客户像看疯子一样的看着我，说：这些组件是别的部门开发的啊！

我说，哪又怎么样呢？虽然不是同一个部门，但是发布的时候总是在一起的，放在统一的持续集成环境可以保证变更的影响最小化：毕竟，本质上CI的作用就是通过频繁的集成来尽早的暴露问题嘛。客户摇摇头，说：版本的CI是不能失败的！我几乎要笑出声来了，说：那要CI干什么，如果它永远不挂的话！客户说：这个是一条开发纪律，版本CI失败了会影响团队的绩效。

也就是说，从某个时刻起，开发部经理被客户逼迫着上线，但由于产品的集成问题，该交付的软件并没有如期交付（当然也没有经过足够的测试）-- 因为没有一个可用的软件包。于是气急败坏的经理在总结经验的时候定下规矩：版本CI不许失败！

这是一个局部优化的典型案例。



全局优化

在遇到问题时，人们总是倾向于解决眼前的问题。而这很可能是“头痛医头，脚痛医脚”的手段，如果缺乏对问题根本原因的分析过程，重复的问题会不断的冒出来，你会为解决这些救火型的而疲于奔命。就像《凤凰项目》中比尔的运维团队一样，团队成员总是被临时插入的

紧急任务打断，而一旦团队将次作为常态，就再也无法从这种没有希望的状态中解脱出来了。团队会变得越来越忙，越来越没有条理。

全局优化强调分析问题的根本原因，从而更加系统的进行调整，而不是做无谓的局部优化。

关注价值流

关于这一点，我事实上已经整理了好几篇博客了：

- 如何持久化你的项目经历
- 不想当UX的开发不是好咨询师

很多时候，程序员很容易深陷技术无法自拔，而忽略了技术服务于业务这个前提。我自己在工作的前几年也是这样，喜欢追求`纯粹`的技术。业务本身可能并不会比技术更有意思，但是使用技术来解决业务中遇到的问题，才是程序员重要的职责。

当然，这种短视并不局限在程序员身上。我看到了很多的趾高气昂的业务人员，对开发人员说：我不管你实现上的细节，只需要你能让我完成X功能和Y功能就行了；也见过抱怨开发自己不自测，把压力push到测试团队的测试人员。

其实，如果将目光放到`价值流`上，这种短视行为就可能得到改善。每个独立的步骤：业务分析，编码实现，测试等都无法带来实际的价值，唯有这个流程串起来才可以。如果团队（包含了业务分析，开发，测试，部署）都聚焦在`价值流`上，才可能实现全局上的优化行为。

在现实中验证和优化

《发布》告诉我们现实世界是多么的混乱和不可控，我们认为的那些小概率事件其实每天都在发生。粗心的程序员写错了日志级别，结果网站上线之后的几天内日志就写满了磁盘；一个未赋值的环境变量导致使用了该变量的脚本执行了危险的操作（`rm -rf $PREFIX/`）；没有为数据库连接设置合适的超时时间而导致系统挂起等等。

这些在开发环境，测试环境，staging环境无法发现的问题，最终会在产品环境中找到你，并对你的数据造成伤害，令你的业务造成损失。

我在很早前的一个项目上，经历过这样一个`有趣`的生产问题，直到今天我还记忆犹新。我为一个C/S架构的文件管理系统设计了一个表结构，由于我们是用C语言来实现这个系统的，所以不得不定义了一些这样的结构体：

```c
typedef struct _RemoteFile{
    char name[128];
    char revision[6];
    //...
} RemoteFile;
```

基于某些原因（可能是序列化库的限制），这里的版本号定义成了一个字符数组，长度为6。这个字段会保存后台的`subversion`返回的版本号。

在投入生产之前，开发和测试人员分别做了大量的测试（好吧，并不是很大量，但是确实做了一些测试）。基本功能都正常，事实上，这个系统良好的运行了3周，第4周的某一天，客户突然抱怨所有文件都无法提交到后台系统了。

VC写的前端应用会报一个类似于`Something Went Wrong`的错误，我们远程查看了后台的日志，发现数据库报了一个`主键冲突`的异常。有经过了1个小时的分析（其实就是远程连接到生产系统，用gdb attach到进行上，然后单步调试），我们发现`RemoteFile`这个结构提的`revision`中的数据赫然是`999999`！。

我们后台的文件管理系统是基于`subversion`开发的，而同一个`subversion`库会共享一个`revision`号：这显然是一个不断自增的数字。在短短的3周内，疯狂的用户们提交了`999999`次，终于在第`1000000`次时保存失败了：事实上，`subversion`已经执行成功了，但是写入数据库时失败了。

反脆弱

如果没有办法避免这种失败（事先要考虑到这么多的异常情况不太现实，而且会投入过多的精力），那么就需要设计某种机制，使得当发生这种失败时系统可以将损失降低到最小。

另一方面，系统需要具备从灾难中回复的能力。如果由于某种原因，服务进程意外终止了，那么一个`watchdog`机制就会非常有用。

反脆弱还包括系统需要提供足够的机制来保护自己，比如`熔断器`模式，`舱壁隔离`模式等等。

`熔断器`模式指当应用在依赖方响应过慢或者出现很多超时时，调用方主动熔断，这样可以防止对依赖方造成更严重的伤害。过一段时间之后，调用方会以较慢的速度开始重试，如果依赖方已经恢复，则逐步加大负载，直到恢复正常调用。如果依赖方还是没有就绪，那就延长等待时间，然后重试。这种模式使得系统在某种程度上显现出`动态性`和`智能`。

`舱壁隔离`模式指将组件部署在不同的应用容器中（独立进程，独立的JVM等），这样避免某些有缺陷的组件将整个系统的资源耗尽， 影响其他的组件正常工作。

在现实世界中，设计一个无缺陷的系统显然是不可能的，但是通过努力，我们还是有可能设计出具有弹性，能够快速失败，从失败中恢复的系统来。

基础设施

传统项目

2012年之前，我在一家传统的软件公司工作。每当我们要新启动一个项目时，项目经费中总会有一项“硬件采购”。其中会描述我们需要多少台服务器（开发环境，测试环境，UAT，生产等），还会描述每个服务器的规格（网卡配置，内存，硬盘等）。当机器购买到之后，我们需要将其部署到机房，接上网线，然后由专门的运维工程师为其配置操作系统，网络，甚至数据库系统。

在开发过程中，如果某个环境由于某些原因崩溃了（比如加班到深夜的程序员睡眼惺忪的执行了 `rm -rf /` ），就只好等运维工程师重新安装。有时候，我们会想想何不把系统做成一个镜像呢？就像虚拟机那样，如果使用虚拟机，每当团队里有人要使用Linux时，我们可以拷贝一份镜像给他。~~但是对于如何将物理机器上的操作系统变成镜像好像也没有成熟的方案，当大家都习惯了这种工作节奏之后，也就不会有人来多问（毕竟多一事儿不如少一事儿）。~~

有了环境之后，开发人员开始写代码，并定期打包，然后部署到各个环境（有时候还要为64位机器和32位机器分别构建不同的软件包）。运维部门有些聪明的家伙会编写一些小脚本，通过 `ssh` 到不同机器来上传应用文件，重启 `J2EE` 容器等，这样他们只需要维护一组IP地址就可以了。

但是现实世界是复杂的，比如开发人员在 `J2EE` 应用中使用了 `properties` 文件来指定数据库的url和文件服务器的地址，这些配置信息在测试环境，staging环境都是不一样的，这样导致的一个问题是：要么我们为不同的环境打不同的包，要么将配置文件放在 `war` 包外边，然后让运维工程师在部署的时候，根据实际情况来修改这个 `properties` ，之后重启容器即可。

当环境慢慢变得多起来的时候，你就可以想像运维工程师的脸色有多难看。程序员最痛恨的就是重复劳动了，运维工程师也一样，谁会喜欢每周三晚上都工作到12点来部署另一个部门的人开发的 `垃圾` 程序呢？

自动化环境搭建

自动化

我们来梳理一下上面这个场景里的问题：

- 开发自己不做部署
- 环境的安装是手工的
- 应用的配置信息需要手工修改

除了艺术品之外，在工业社会里，手工就意味着低效，容易犯错，且不可持续。一切可以自动化的，都应该被自动化起来。

一个软件系统往往会包含很多的组件（消息队列服务，应用服务器，数据库服务器，负载均衡器，反向代理，文件服务器等），而且每一套环境（开发环境，测试环境，UAT，Staging，生产）还有自己独立的组件。

因此一个操作系统要被配置成系统的某个组件还需要做很多工作：以Java为例，我们需要安装特定版本的JDK，设置 `CLASSPATH` 环境变量，修改操作系统的内核参数，创建特定用户（数据库用户等），修改一些目录的权限等等。这些操作如果交给人工来完成，必然会出现各种错误（想想这个过程要被在不同的环境中重复多遍，出错的机率会大大增加）。

事实上业界已经有了很多帮助开发/运维工程师进行环境安装的工具，比如

- Chef
- Puppet
- Ansible

前两者我已经在《轻量级Web应用开发》有过介绍，这里我们以 `Ansible` 为例来描述。

Vagrant

`Vagrant` 提供对虚拟机的封装，使用它可以很容易的通过配置的方式来定义一个 `虚拟机` 。

使用 `Vagrant` ，你只需要定义一个文本文件 `Vagrantfile` 即可。 `Vagrant` 自带的命令行工具 `vagrant` 会尝试加载这个文件，并按照其中的配置来启动虚拟机。 `Vagrantfile` 按照 `ruby` 的语法编写，不过不用担心，你无需在其中定义函数或者类，只需要做一些配置即可。

下面是一个简单的虚拟机定义：

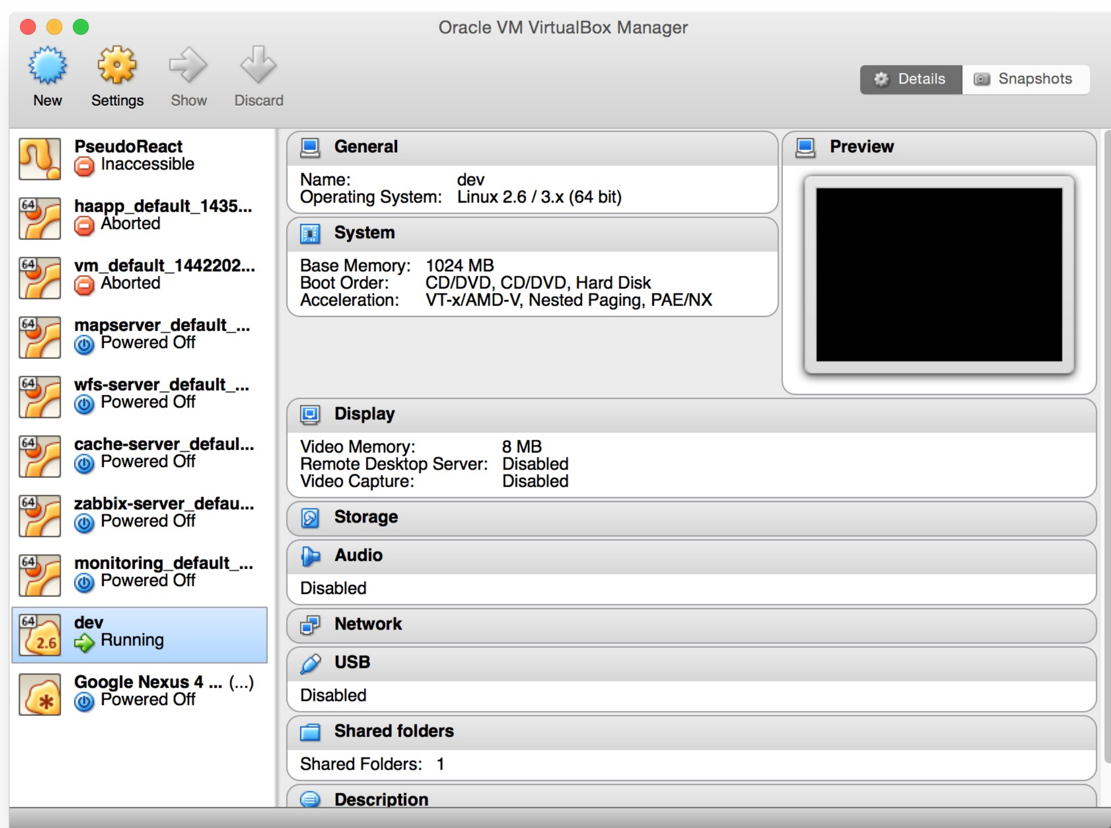
```
Vagrant.configure("2") do |config|
  config.vm.box = "precise64"
  config.vm.network "private_network", :ip => "192.168.2.100"
end
```

我们指定了虚拟机使用 `precise64` （precise是一个ubuntu的发行版，64表示它是一个64位系统的镜像）这样一个镜像，并且给这个虚拟机分配一个私有的IP地址，这样我们就可以在宿主环境中通过这个IP来访问该虚拟机了。

定义了 `Vagrantfile` 之后，使用 `vagrant` 工具的子命令就可以启动虚拟机了

```
$ vagrant up
```

你可以在 `VirtualBox` 的界面里看到正在运行的虚拟机（Vagrant在底层使用了VirtualBox的虚拟机，而不是自行开发另外一套）：



启动之后，你可以通过

```
$ vagrant ssh
```

来登录到虚拟机中。

如果你不知道使用哪个镜像，不知道如何配置 `Vagrantfile` ，可以使用这个命令来从头开始：

```
$ vagrant init hashicorp/precise64
$ vagrant up
```

`vagrant` 命令会自动下载镜像，并设置环境，然后启动虚拟机。

在工程实践里， `Vagrantfile` 会checkin到代码库中，这样团队里的其他人也可以很容易的在本地重新搭建相同的环境。另外，我推荐你将 `box` 的版本尽量和生产环境一致（比如都使用ubuntu的precise64位），这样可以尽早发现一些环境相关的问题。

初始化环境

`Vagrant` 还提供了丰富的机制来初始化环境。你可以使用简单的 `Shell` 脚本，或者全功能的 `Ansible` ， `Chef` 等来初始化环境。

设想你需要在虚拟机环境就绪后，在 `vagrant` 用户的home目录下创建一个叫 `workers` 的目录，安装一个叫 `wget` 的软件包，然后下载一个网络上的文件到 `workers` 目录。

要完成这样的动作，我们可以在当前目录（和 `Vagrantfile` 放在一起）创建一个 `setup.sh` 的脚本：

```
#!/usr/bin/env bash

mkdir -p ~/workers
sudo apt-get update
sudo apt-get install wget
wget http://host:port/resource.zip -O ~/workers/resource.zip
```

然后在 `Vagrantfile` 中加入：

```
Vagrant.configure("2") do |config|
  config.vm.box = "precise64"
  config.vm.provision :shell, path: "setup.sh"
end
```

当 `Vagrant` 在初始化虚拟机的时候，会执行 `setup.sh` ，这样我们就得到了一个经过设置的环境。设置环境可能会是一个非常复杂的过程，比如安装web服务器，定义缓存目录，安装监控服务器的客户端，为某些应用程序创建专用用户，修改权限等等，如果用shell来写，会比较复杂。

`Vagrant` 支持很多的 `provision` 的工具，比如 `Ansible` 来完成这种复杂的操作。

Ansible

Ansible是一个自动化配置工具，相对于 `Chef` ， `Puppet` ，它的安装和配置更加简单（无需在被配置的服务器安装额外的Agent程序）。它通过 `ssh` 将一些Ansible模块部署到远程机器上，然后执行。

使用 `Ansible` 可以同时配置，更新多个机器。目前很多企业都会使用各种各样的云产品，比如AWS的EC2，阿里云等，通过 `Ansible` 可以很容易的将这些环境配置变成自动化。在企业内部的私有云（从一台服务器划分出来的众多虚拟机）中，也可以使用 `Ansible` 来减少配置环境的时间，提高效率。

在一个冬日的下午，我和一个新手程序员结对在服务器上修改 `tomcat` 服务器的一些日志的配置，折腾了很久之后，我放弃了。我心想，要不删了 `webapps` 这个目录重新部署一下看看吧，可能是缓存问题也说不定。不过，头昏脑涨的我并没有发现敲入的命令是 `rm -rf /usr/share/tomcat7` 。新手程序员问我， `rm -rf` 是什么意思？我一边用力的敲下回车键，一遍警告这个新手：“rm -rf是一个非常危险的操作，它表示要强力删除整个......”。等我发现我删除的是 `tomcat7` 的时候已经太晚了，我们的QA环境彻底挂了，所有人都被block住了（还好不是在其他人给客户showcase的时候）。

另一个工程师，我们姑且称之为 `运维工程师` 吧，花费了好几个小时来重新安装 `tomcat` ，以及其中的各种 `jvm` 参数。

我们在随后的几周里，引入了 `Ansible` ，这样即使头昏脑涨的程序员无意识的敲入了愚蠢而致命的命令也无所谓，我们只需要2分钟就可以配置好一个 `tomcat` 服务器，崭新的。

惯例

`Ansible` 中的一些关键概念：

1. role 定义一个角色，比如nginx就可以是一个角色，要完成nginx的安装需要很多小的步骤，这些步骤都包含在nginx这个role中
2. inventory 定义一组环境，比如Web服务器需要三台做负载均衡，数据库由两台服务器组成等，这些都可以通过inventory文件来描述，inventory文件被称为清单文件
3. playbook 定义在哪些inventory应用哪些role

`Ansible` 中有一些惯例，遵循这些惯例有助于你快速读懂其他人写的 `playbook` / `role` 。

```
production              # 生产环境的清单文件
staging                 # staging环境的清单文件
qa                      # 测试环境的清单文件

site.yml                # 主playbook
webservers.yml          # web服务器的playbook
dbservers.yml           # 数据库服务器的playbook

roles/
    common/             # this hierarchy represents a "role"
        tasks/          #
            main.yml    # 具体任务定义
        handlers/       #
            main.yml    # 回调任务
        templates/      #
            nginx.conf.j2 # 模板文件
        files/          #
            app.conf    # 需要拷贝到被配置环境中的文件
        vars/           #
            main.yml    # 变量定义
        defaults/       #
            main.yml    # 低优先级变量定义
        meta/           #
            main.yml    # 元数据，用以表述作者信息，定义依赖等

    webtier/            # 另外一个role，结构和`common`一致
    monitoring/         # 用于监控的role，结构同上
```

比如一个简单的 `inventory` 文件看起来是这样的：

```
[webservers]
10.29.2.1
10.29.2.2
10.29.2.3

[dbservers]
10.29.2.4
10.29.2.5
```

没错，它就是一个简单的ini文件。如果你需要添加新的机器，只需要将域名/IP地址添加到对应的小节即可。

`Ansible` 使用yml作为配置，我们来看一个playbook的例子:

```
-   name: webservers
    hosts: webservers
    roles:
        - roles/nginx
    user: robot
    sudo: true

-   name: dbservers
    hosts: dbservers
    roles:
        - roles/mongodb
    user: robot
    sudo: true
    environment:
        http_proxy: http://user:pass@proxy.host:8080
        https_proxy: http://user:pass@proxy.host:8080
```

这个playbook定义了两个环境的配置信息：`webservers` 和 `dbservers` 。`webservers` 中的所有主机会被应用 `nginx` 角色（安装和配置nginx，并启动nginx服务），而 `dbservers` 中的主机会被应用 `mongodb` 的角色。

在 `dbservers` 中，我们还加入了 `environment` 节，其中定义了可以用在安装过程中的一些环境变量设置。

定义好之后，你可以通过下列命令来执行这个 `playbook` ：

```
ansible-playbook -i qa playbook.yml
```

命令

`Ansible` 内置了很多常用的命令来简化配置的工作，比如安装软件包，拷贝文件，使用模板，创建用户，创建目录等。

安装软件包

```
- name: install package
  apt: name=nginx state=present
```

`apt` 命令可以用于安装一个软件包，它相当于在主机上执行 `apt-get install nginx -y` 。如果你需要安装多个软件包，可以采用 `with_items` 子命令：

```
- name: install packages
  apt: name={{ item }} state=present
  with_items:
      -  nginx
      -  python
      -  git
```

创建目录

```
- name: create directory
  file: path=/home/vagrant/workers state=directory
```

拷贝文件

```
- name: copy file to workers folder
  copy: src=resource.zip dest=/home/vagrant/resource.zip
```

如果你要使用的命令，正好 `Ansible` 没有内置，你还可以使用 `command` 命令来执行：

```
- name: universal command
  command: ls -alt /home/vagrant
```

如果要完成 `Vagrant` 小节中的例子，我们的配置看起来就是这样的：

```
- name: create directory
  file: path=/home/vagrant/workers state=directory

- name: install package
  apt: name=wget state=present

- name: download file
  command: wget http://host:port/resource.zip -O ~/workers/resource.zip
```

当然，根据预定，我们会把变量放在 `vars` 目录的 `main.yml` 中，比如上面 `apt` 例子中的多个包的安装，我们会在 `vars/main.yml` 中定义变量

```
packages:
      -  nginx
      -  python
      -  git
```

然后在 `tasks/main.yml` 中引用：

```
- name: install packages
  apt: name={{ item }} state=present
  with_items: '{{ packages }}'
```

独立使用

通常我们会在一个Linux环境安装 `Ansible` ，这个环境专门做环境初始化的工作。如果你使用ubuntu环境，可以通过预编译的二进制包来安装：

```
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

当然，你也可以通过源码来安装：

```
$ git clone git://github.com/ansible/ansible.git --recursive
$ cd ./ansible
$ source ./hacking/env-setup
$ sudo make install #安装到系统路径，其他用户也可以使用
```

安装完成之后，你会得到 `ansible` 命令和 `ansible-playbook` 命令。

在执行 `ansible` 命令之前，我们需要定义 `ansible` 对应的远程机器列表配置。你需要在 `/etc/ansible/hosts` 文件中添加所有需要配置的机器IP地址或者域名（如果没有这个目录和文件，可以直接创建）。

文件内容就是每个地址一行的形式：

```
10.29.2.1
10.29.2.2
10.29.2.3
```

`ansible` 命令可以用来向inventory执行shell命令，比如：

```
$ ansible all -m ping -u robot -b --become-user root
```

上面的命令会向主机 `all` （ `/etc/ansible/hosts` 文件中指定的所有地址），执行一个 `Ansible` 模块 `ping` （-m ping），使用用户 `robot` （-u robot），并且以另一个用户身份 `root` （-b --become-user root）。

如果没有设置过 `ssh` 的私钥，还需要指定 `--ask-pass` 选项，否则 `ansible` 会尝试用ssh登陆，然后失败。

```
$ ansible all -m ping -u robot -b --become-user root --ask-pass
```

除了使用内置模块之外，你开可以使用其他任何shell命令：

```
$ ansible all -a "/bin/echo hello" -u robot --ask-pass
```

`Ansible` 会直接在远程机器上执行对应的shell命令。

在Vagrant中使用

`Vagrant` 可以很容易的和 `Ansible` 集成在一起，只需要指定 `config.vm.provision` 为 `ansible` 即可：

```
Vagrant.configure("2") do |config|
  config.vm.provision :ansible do |ansible|
    ansible.playbook = "playbook.yml"
  end
end
```

当然，你可以定义多个虚拟机，然后并发的来自动化配置，比如像这样：

```
(1..5).each do |machine_id|
  config.vm.define "machine#{machine_id}" do |machine|
    machine.vm.hostname = "machine#{machine_id}"
    machine.vm.network "private_network", ip: "192.168.2.#{20+machine_id}"

    if machine_id == N
      machine.vm.provision :ansible do |ansible|
        ansible.limit = "all"
        ansible.playbook = "playbook.yml"
      end
    end
  end
end
```

容器技术与Docker

和 `微服务` 一起火起来的，还有容器技术。通过容器技术，应用程序可以和环境打包在一个包中，然后运行在任何支持容器的操作系统中，容器隔离了所有的物理层。这使得人们非常轻量级地、安全可靠地发布软件包（包含了应用程序和运行时环境）。

传统意义上，人们在持续发布流水线上编译，测试，构建出一个软件包，然后再自动化配置一套环境，再将软件部署其上。现在人们可以很容易的为流水线添加一个新的job，使其构建出一个独立的容器。这个容器可以直接运行在像AWS这样的云提供商的机器上。

Docker是目前最为流行的容器技术，它可以运行在主流的Linux主机上（内核版本3.10以上）。通过Docker Toolbox，你还可以在Mac下，甚至Windows上运行Docker（当然，只是Docker的客户端，但是即使这样也可以获得Docker带来的好处）。

如果你在 `Mac` 下工作（如果在Linux下工作，按照官方的指导即可，如果在Windows下工作，那我建议你还是换成Linux或者Mac吧），可以安装 `Docker Toolbox` 来使用Docker。 `Docker Toolbox` 中包含了一组工具，包括：

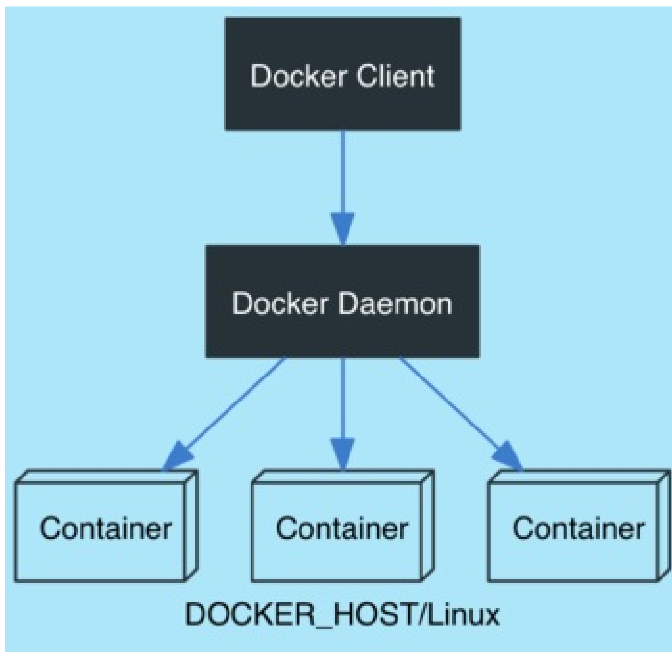
- docker-machine 应用程序
- docker-compose 应用程序
- Kitematic，docker的图形化客户端
- VirtualBox
- 一个包含了Docker所需配置的shell环境

概念

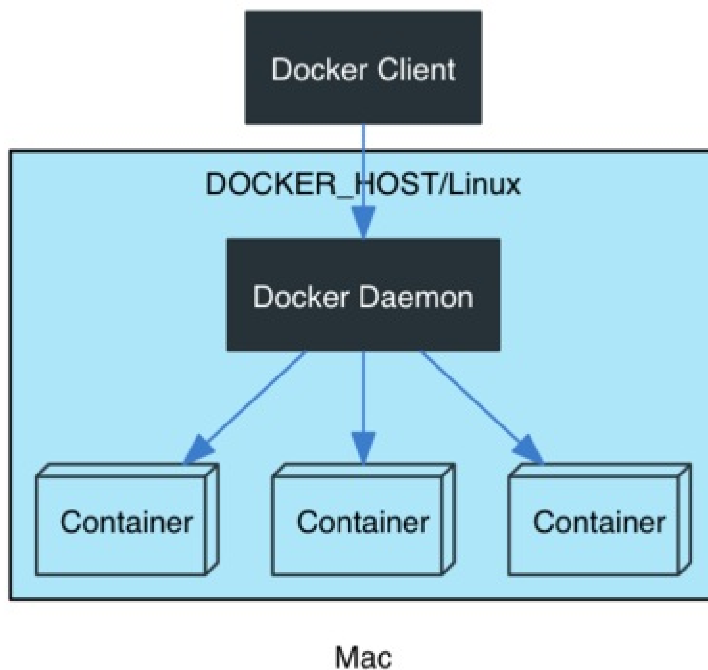
事实上，我们通常说的Docker分为三个部分：

- docker daemon（只能运行在Linux下，使用了Linux的一些内核特性）
- docker client（客户端程序，可以运行在Linux/Mac/Windows下）
- container（容器，包含了应用程序和其依赖的环境）

在Linux环境中，这三者都在同一个环境中：



而在Mac和Windows中，docker client运行在宿主环境中，docker daemon和container都在一个Linux的虚拟机中：



docker-machine

`docker-machine` 可以用来创建和管理Linux虚拟机，这些Linux虚拟机具备了运行Docker Daemon的条件。宿主操作系统的客户端可以连接到该Linux虚拟机。

你可以使用下面的命令来创建一个虚拟机：

```
$ docker-machine create --driver virtualbox myvm
```

使用：

```
$ docker-machine ls
```

来查看已经运行的 `docker` 虚拟机：

```
$ docker-machine ls
NAME    ACTIVE    DRIVER       STATE     URL                          SWARM
dev               virtualbox   Running   tcp://192.168.99.100:2376
```

可以看到，我本地有一台名为 `dev` 的虚拟机，状态为 `运行中` 。如果这个机器没有启动（STATE状态部位Running），你可以通过：

```
$ docker-machine start dev
```

来启动它。启动之后，你可以使用 `env` 子命令来查看机器的配置

```
$ docker-machine env dev
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/jtqiu/.docker/machine/machines/dev"
export DOCKER_MACHINE_NAME="dev"
# Run this command to configure your shell:
# eval "$(docker-machine env dev)"
```

事实上，你可以创建多个 `docker` 虚拟机，因此在你的shell中执行下列命令来设置环境变量：

```
eval "$(docker-machine env dev)"
```

这样，你本地的环境变量如 `DOCKER_HOST` ， `DOCKER_MACHINE_NAME` 等就会被设置为虚拟机 `dev` 对应的值，后续的所有docker客户端命令就会在该机器上生效。

启动了 `docker` 虚拟机之后，你就可以使用 `docker` 客户端程序来启动container了

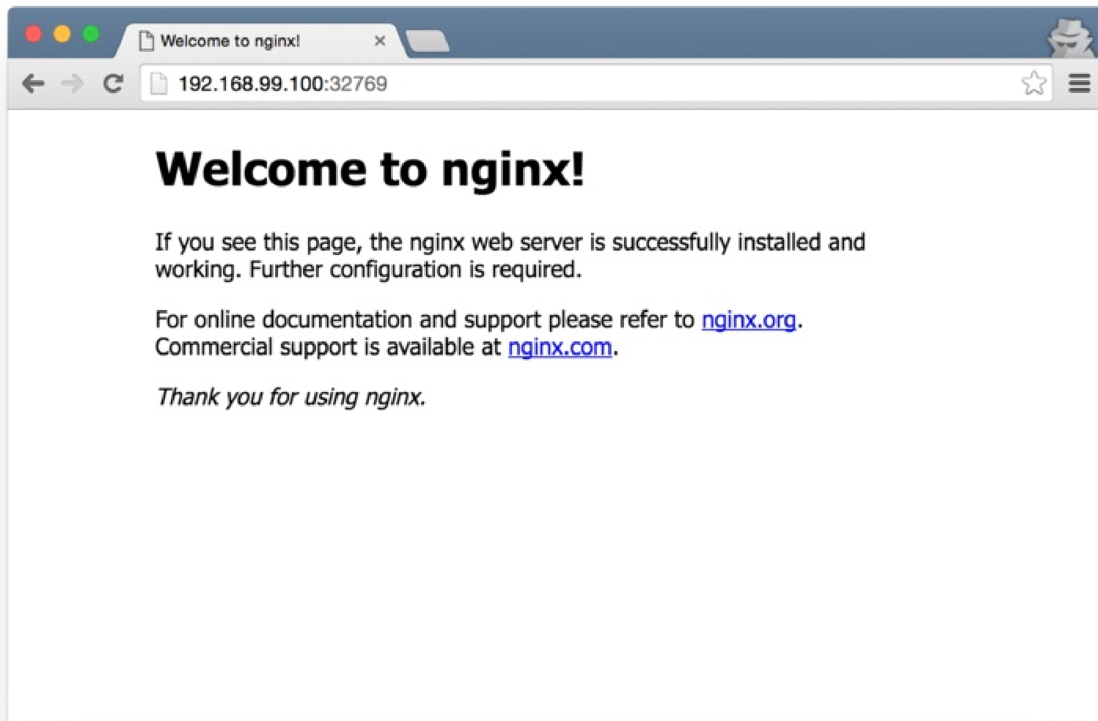
```
$ docker run -d -P --name web nginx
```

这条命令会使用 `nginx` 这个docker image来启动一个容器，容器的名字叫 `web` 。默认的docker在执行完命令之后会退出， `-d` 选项使其进入后台服务模式，即不退出。 `-P` 选项会将容器中的端口暴露给宿主环境。

如果你本地没有 `nginx` 镜像，docker会尝试从远程的中心仓库下载该镜像，最后启动：

```
$ docker run -d -P --name web nginx
Unable to find image 'nginx:latest' locally
Pulling repository docker.io/library/nginx
c0e6aba9c87a: Pulling dependent layers
...
```

启动之后，你就可以通过 `DOCKER_HOST` 对应机器的80端口来访问该nginx实例了：



docker-compose

非功能需求

在软件开发过程中，除了功能需求（用户可以购物，可以下订单等）之外，还有很多需求需要确定。这些需求与具体的业务需求没有关系，却对整个系统有着十分重要的影响。

这里略举几个例子：

- 安全性
- 吞吐量
- 负载
- 响应速度
- 日志记录
- 备份方案（归档）
- 可扩展性（横向扩展，水平扩展）

每个业务系统都或多或少需要涉及这些`非功能需求`，但是各个系统的要求有不尽相同。一个网上银行系统，和一个企业内部的知识管理系统对安全性的要求就完全不一样。业务面向全球的电商网站，和西安市的驾校预约网站对吞吐量和响应速度的要求也不会一样。

在梳理了业务方面的需求之后，我们还需要对非功能需求进行深入的讨论和了解，这在我们做技术选型时会有巨大的影响。

监控

- 应用状态
- 系统状态

Metrics

```
dependencies {
    compile 'org.springframework:spring-context:4.2.4.RELEASE'
    compile 'org.springframework.data:spring-data-mongodb:1.8.4.RELEASE'

    compile 'io.dropwizard.metrics:metrics-core:3.1.2'
    compile 'io.dropwizard.metrics:metrics-jvm:3.1.2'
    compile 'io.dropwizard.metrics:metrics-graphite:3.1.2'
    compile 'com.ryantenney.metrics:metrics-spring:3.1.3'
}
```

```
@Configuration
@EnableMetrics
public class MetricsConfig extends MetricsConfigurerAdapter {

    @Override
    public void configureReporters(MetricRegistry metricRegistry) {
        Graphite graphite = new Graphite(new InetSocketAddress("192.168.99.100", 2003)
);
        GraphiteReporter graphiteReporter = GraphiteReporter.forRegistry(metricRegistr
y)
                .prefixedWith("juntao-laptop")
                .convertRatesTo(TimeUnit.SECONDS)
                .convertDurationsTo(TimeUnit.MILLISECONDS)
                .filter(MetricFilter.ALL)
                .build(graphite);

        registerReporter(graphiteReporter);
        graphiteReporter.start(1, TimeUnit.MINUTES);


        metricRegistry.registerAll(new MemoryUsageGaugeSet());
        metricRegistry.registerAll(new ThreadStatesGaugeSet());
    }

}
```

```
@RestController
public class PersonController {
    @Autowired
    private PersonRepository personRepository;

    @Metered(absolute = true, name = "metered.people.get.all")
    @Timed(absolute = true, name = "people.get.all")
    @RequestMapping(value ="/people", method = RequestMethod.GET)
    public List<Person> findAll() {
        return personRepository.findAll();
    }
}
```

Graphite

Grafana

collectd

技术选型

2011年，我们项目组在开发一个给公司内部使用的`IDE`，这个IDE是一个典型的C/S结构的软件，客户端使用VC编写，服务器端则运行在`Linux`上，所有的编译，链接，模拟运行等都在服务器上，客户端只提供一个简单的文本编辑器。

客户端与服务器通过一系列的服务来通信，这些服务有点像当时比较流行的SOAP方式，但是又不完全是（比如没有服务发现）。比如`创建文件`，`编译模块`，`测试模块`都是不同的服务。客户端通过HTTP来调用这些不同的服务，来提供完整的功能。

截止目前，听起来还挺正常吧？问题是，后台使用了C作为主要编程语言，数据库访问使用Oracle提供的ProC（那确实是个灾难，你应该可以想象在C语言中嵌入那些奇形怪状的预处理代码有多恶心），再往后是TUXEDO中间件。

```
//这个例子来自Oracle ProC官方网站
//http://docs.oracle.com/cd/E11882_01/appdev.112/e10825/pc_08arr.htm#g20885

int   emp_number[20];
float salary[20];

EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT empno, sal FROM emp;

EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND do break;
for (;;)
{
    EXEC SQL FETCH emp_cursor
        INTO :emp_number, :salary;
    /* process batch of rows */
    ...
}
...
```

我们项目中有6个开发，3个测试，项目进行了差不多1年，仍然没有一个比较稳定的版本。后来部门决定换掉那个老旧的`TUXEDO`，换成另一个自己写的用C语言编写的中间件。这个工具后来在深圳分公司做试点，但是依旧不是很稳定。通过这个IDE开发出来的服务很难调试，我们不得不将对`gdb`进行了部分封装，然后尝试使用`netbeans`协议来做远程的debug，不过过程中发现这种机制需要编写很多代码，而且这部分代码则更加难以调试。

一个折中的办法是让开发通过`ssh`远程登录到服务器上，用`gdb` attach到对应的进程上进行调试。不过这个由于IDE的设计初衷相违背（这个IDE的目的是降低服务的开发难度而不是增加）。印象中，在我离开那个项目之前，这个问题都没有得到很好的解决。

这是一个典型的，被技术选型害了的场景。如果我们使用更易于开发，调试，维护的Java平台，或者动态语言如Ruby，Python，开发周期至少可以缩短一半。至于设计时考虑的那些性能问题，其实很可能并不是真正的问题，我们并不需要支持数万人同时在线，也无需考虑超大规模的数据库备份方案，毕竟当时整个公司的开发也不过200人。

我们选择了错误的编程语言，错误的数据库模型，错误的软件架构，实现了一个没有太多人使用的"产品"。其实归纳起来，成语 `方枘圆凿` 可以很传神的描述我们做的事情。

技术选型



整体架构

- 单机版桌面应用
- C/S
- B/S
- B/S+
- 部署模型（Web服务器，应用服务器，负载均衡，缓存，DNS）

前端

- 浏览器兼容性
- 响应式设计
- 移动端网站
- 前端模板
- CSS、JS预处理器
- 前端框架选择
- 测试工具

移动端

Native

Hybird

服务器端

- 性能要求
- 交付时间
- 业务模型（动态模型NoSQL）
- Web容器

- 通信协议
- 构建工具
- Web框架
- 测试工具

数据存储

关系数据库

NoSQL

项目保证

- 源码管理
- 持续集成
- 环境搭建
- 仓库

可持续的软件架构

一方面，我们的确需要灵活的，易于扩展、容易修正的框架。另一方面，我们需要这些基础框架尽可能的坚固，以支撑构筑于其上的应用的稳定性。

为了灵活性我们会提供很多的配置，选项（比如定义一个简介层来适配不同的数据库类型），这样我们在后续的开发中如果需要切换到另一个实现上时仅需要付出很小的代价即可。但是，往往我们并不能预测变化会在哪个维度发生，也许数据库从 `MySQL` 切换到了 `Oracle` ，也许是Web容器变成了Websphere，又或者模板引擎从 `Freemark` 变成了 `Velocity` 。

由于我们在预测未来上做的非常糟糕，事实上很难在开始时就预料到变化的方向。

方法论

瀑布模型

敏捷与精益

周围经常有很多人在争论到底那种开发方式是正统的敏捷，就好比华山派的剑宗，气宗之争一样，只不过参与争论的门派更多而已。初学者会认为迭代开发，TDD，结对编程，站会就是敏捷；更高级一些的则又认为这些实践层面的未必是敏捷，敏捷更多的是根据实际情况来调整策略，让团队自己管理自己的流程。

当然，软件开发的各种方法论发展到今天，可能很难看到某个组织或者团队在进行某种纯敏捷或者纯瀑布的开发方式了，各种实践你中有我，我中有你，互相影响。

比如在ThoughtWorks，我们很多团队在传统的Scrum中融入了一些看板方法。除了站会，迭代开发之外，团队引入看板，控制在制品数量，价值流驱动等等。而有些客户团队里，则是在敏捷开发的形式下进行瀑布模型的开发，团队也进行站会，也划分迭代，但是仍然有独立的测试部门，运维部门，部门墙森然林立，任何一个单独的团队都无法描述一个完整的价值流，没有人真正知道一个功能的周期时间如何，也没有人真正了解整个流程（需求从何而来，如何做系统测试，如何保证最后的部署）。

追本溯源

敏捷一词来源于2001年初美国犹他州雪鸟滑雪圣地的一次敏捷方法发起者和实践者的聚会，与会者总结出一套软件开发价值观。

```
我们一直在实践中探寻更好的软件开发方法，
身体力行的同时也帮助他人。由此我们建立了如下价值观：

个体和互动 高于 流程和工具
工作的软件 高于 详尽的文档
客户合作 高于 合同谈判
响应变化 高于 遵循计划

也就是说，尽管右项有其价值，
我们更重视左项的价值。
```

你会发现几乎所有的这些实践都很难直接和我们平时说的“敏捷”对应起来。所以，所谓谁对谁错压根没有意义。因为敏捷只是一些理念，还没有具体到可以实践的层面。

所幸人们开发了一些工具/实践，来支持这样的理念，流传最广的是极限编程和**Scrum**两类。后面的看板方法则是在精益的基础上扩展出来的。

精益原则

与敏捷宣言诞生于软件开发领域不同的是，精益的概念来自于制造业，具体来说，是日本丰田公司。该方法帮助丰田成为世界上最成功的汽车制造商，实践证明，精益制造原则同样适用于软件开发领域。

根据精益制造原则，人们为软件开发也总结了如下`七条原则`：

- 消除浪费
- 增强学习
- 尽量延迟决定
- 尽快发布
- 下放权力
- 嵌入质量
- 全局优化

极限编程（XP）

极限编程是敏捷中应用非常广泛的方法学之一，

- 现场客户
- 结对编程
- 测试驱动开发
- 代码所有权共享
- code review

Scrum

- 迭代开发
- 每日站会
- 回顾会议（retro）
- 用户故事
- backlog
- 跨功能团队

看板方法

热力学第二定律的一个推论是：如果没有外部能量输入，封闭系统趋向越来越混乱。我们需要小心翼翼的维持一个系统（不论是软件系统，还是某个地区生态系统），使其达到某种程度的平衡。而且这个过程是动态的，需要不断调整的。

我们生活在其中的这个物理世界，是一个充满了随机，不确定，不完美的环境，所有一切顶多只能达到“刚好够用”的程度，一个细小的环境变化都可能使整个系统坍塌，崩溃。我们的呼吸系统非常适合20%的含氧量，一旦低于这个值，我们身体的各个子系统就开始出现各种问题，甚至宕机。我们的骨骼强健到足以支撑我们跳跃，奔跑，追击猎物，和不算太大的肉食动物搏斗，但是它并没有强健到缓冲从3层楼跳下来的冲击力。

我们一直引以为豪，或者至少觉得不可思议的思维系统，也是如此。平时大脑中的各种想法纷至沓来，而在我们真正需要的时候（比如公共演讲开场时），它又会变成一片空白。记忆破碎，短暂，而且它会发生偏差，遗漏，即使很重要的事情。

不论我们是如何认为的，人们在做出决策时，事实上是充满随机性和非理性的。即使我们认为非常聪明，非常有经验的那些人也无法幸免。做决策时，大脑很容易被一些事实上并不重要的因素影响，并形成不准确的判断。

既然如此，我们又如何敢将希望寄托在这些“不靠谱”的硬件上呢？

另一方面，人们热衷与做计划，然后按部就班地去实施。

~~传统的那些失败项目中，人们制定大而全的计划~~

中国有句古话，叫“凡事预则立，不预则废”，强调了做事情时制定计划的重要性。短期的计划当然是有用的，而且是关键的。

</del>

现实世界里，人们在实施软件工程时，通常有两个流派：计划派和无计划派。

计划派

计划派无一例外的喜欢大而全的计划，相信实现者会按照这个计划按部就班的实施，设计，开发，测试，部署，最后人们准确地在发布日饮酒庆祝上线成功。但是等一下，这种事情真的发生过吗？

计划派忽略了工程中各个步骤之间的交互，比如开发阶段发现设计有问题，测试阶段发现开发有问题等，这种异常的流程会导致之前的计划变形，而这些异常情况简直就是每个工程中正常部分。毕竟，谁能确保设计文档中的提到的消息队列服务在开发中一定可以实现呢（特别是只有很短工期的情况下）？

结果就是，每个部门都尝试为自己的环节加上缓冲，这会导致整个工程的时间变长。投资方（比如业务部门）则会根据经验压缩这个时间（毕竟，历史上IT部门从来没有兑现过自己的承诺嘛），或者加入额外的需求。这些都会导致项目的延期，或者功能的不完善。

无计划派

无计划派则是迫于压力，每日疲于应付各种突发情况，或者工作优先级随时都可能调整，一切以“高优先级”的客户需求为准。但是其实谁也不知道下一个任务会在合适插入你的待办事项列表。

2010年，我就在这样一个团队工作，开发人员每天扮演的就是救火队的角色。我们的工作可以分为两类：修改之前由于赶进度而产生的遗留问题，或者在赶进度。事实上，也并没有什么所谓的进度，我们有一个待办事项列表，列表中的条目有时还互不相关（就好像JavaScript中的数组一样，你可以在其中存入各种类型的数据）。

- 工作流系统中DSL部分的测试（有人抱怨有语法错误）
- 自定义文件管理系统（客户说必须在一周后可用）
- 修改部署脚本，可以方便运维部门的人自动安装
- 为某个Web服务器启动gzip

而总有一些优先级更高的任务被插入，这些任务可能来自于某个重要客户的一个暗示（是的，暗示！还有比猜测客户的真实需求更不靠谱的事儿吗？），也可能是公司某个其他领导的一个讲话。

另一个流派

我在这两种流派的公司都待过，也目睹了在这种情况下软件项目是如何失败的。幸运的是，除了这两个流派之外，我还见过其他的流派：粗粒度的规划+短期的计划。

项目实践

我的最近的一个项目中，客户想要我们在4周之内搭建出一个原型出来，通过这个原型，客户想要看到一些业务场景在可以通过Web化的方式来实现（之前是一个单机版的应用）。这个项目的挑战在于搭建原型的时间很短，我们需要做一些取舍：比如哪些实践是必须的，哪些不是。

虽然有一些压力，但是我们还是坚持了自己的想法。一开始就搭建了CI环境，甚至有了自动部署到测试环境的构建任务，开发的每一次提交都会触发一次自动部署。

项目开始还算顺利，在第迭代1的第二周，我在调试一个异步的组件：这个组件从消息队列得到通知，然后解压用户上传的文件，紧接着调用另外一个服务做一些数据转换，最后再将转换过的数据保存到数据库。我们很快的为这个组件创建了新的版本库，新的CI构建，以及新的自动化部署脚本。

由于本地的开发环境是Windows，我们很难模拟一套完整的系统（该死的Windows下的路径名），所以我们决定在一个Linux机器上做测试。有自动化部署这套机制的保证，我们可以非常快速的进行系统的集成测试：消息的生产者发送消息给队列服务器，队列再分发到我们的组件上。

有一天下午，有人在服务器上修改了用户权限（好吧，那个人就是我了），然后所有的自动化部署都失败了（由于 `ssh` 的鉴权问题），团队里的其他人不得不手工的进行部署：本地提交代码之后，等CI服务器运行完所有测试之后，开发者手工的触发一次部署命令，同时输入密码。

本来我想着这个过程不会那么频繁，但是事实上它比我想象的要频繁的多。本地修改几行代码，commit一次，手工部署，测试，发现问题，再来一次。在我崩溃前，另一个同事也终于受不了了，他登录到服务器上，修改了ssh配置文件的权限，一切又恢复了正常。

这个故事除了证明“恶心的事情频繁做”这个原则之外，也说明了一个持续集成，持续发布的机制是多么重要。即使是在开发环境、开发场景中也是如此。

想想你可以让一个新特性在 `10分钟` 内上线是一种什么样的体验！

服务器端应用的持续交付

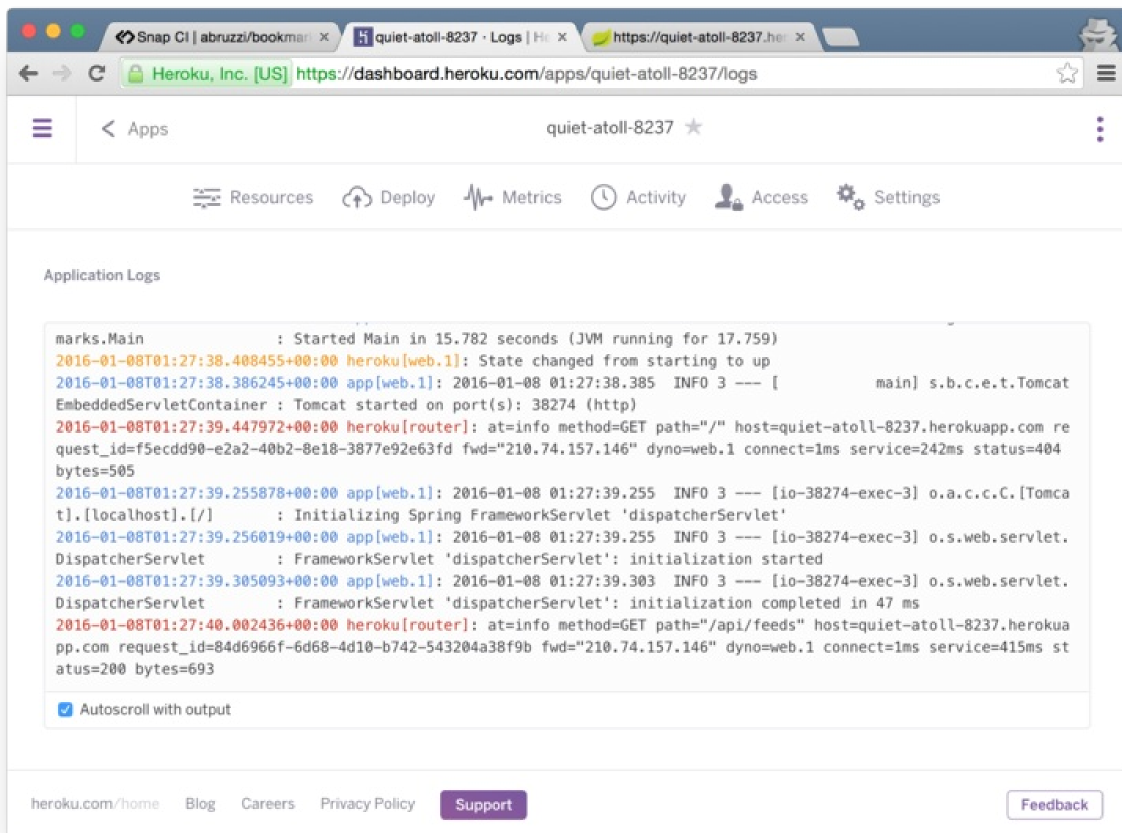
本文将使用一些免费的服务来为你的项目搭建 `持续交付` 平台，这些服务包括

- 持续集成环境
- 持续部署环境
- 服务端应用托管

以及一些可以用于本地开发使用的开源工具如：

- 基于Node的构建monitor
- Heroku的命令行工具
- Travis CI的命令行工具

除此之外，我们在过程中编写的脚本还可以用以本地构建，如果你的团队中正好已经有CI工具/CD工具，将这些脚本集成进去也是一件非常容易的事情。



背景知识

软件的度量

传统的管理方法论，在软件开发这个领域来说基本上是不工作的。软件项目的不确定性使得人们畏惧，管理者希望通过一些数字，指标来让自己感到某种虚幻的“掌控感”。软件行数，测试覆盖率，代码故障率等数字的名声基本上已经很糟了，经常有人拿来讽刺那些追求虚幻掌控感的“领导”。

但是有一个数字，即使最顽固的“自由主义者”也会认为是有意义的，那就是周期时间（cycle time）。简而言之，就是一个需求从产生到最终上线所需要的时间。其中包括了需求分析，设计，编码，测试，部署，运维等活动，可能还会包含后续的监控。

其实不论是瀑布模型，还是迭代开发的方式，或者其他的方法论，周期时间的缩短都是至关重要的。而具体到周期内，单纯的开发时间变长或者测试时间变长都无关紧要。比如项目A的开发时间是测试时间的2倍，项目B则恰恰反过来，这并不能说A做的比B好，真正有意义的是A的周期时间是否比B更短。

单纯改善项目过程中的某一个阶段的时间，可能并不能达到预期的目的。局部优化并不一定会带来全局的优化。换言之，通过某些策略来提高软件测试的效率未必能减少周期时间！。

持续交付

传统情况下，企业要进行软件开发，从用户研究到产品上线，其中会花费数月，甚至数年（我的一位印度同事给我聊起过，他的上家公司做产品，从版本启动到版本上线需要整整两年时间！）。而且一旦软件需求发生变更，又有需要数月才能将变更发布上线。除了为变更提交代码外，还有很多额外的回归测试，发布计划，运维部门的进度等等。而市场机会千变万化，在特定的时间窗口中，企业的竞争者可能早已发布并占领了相当大的市场份额。

在软件工程领域，人们提出了持续交付（continuous delivery）的概念，它旨在减少周期时间，强调在任何时刻软件都处于可发布状态。采用这种实践，我们可以频繁，快速，安全的将需求的变化发布出来，交由真实世界的用户来使用，在为用户带来价值的同时，我们也可以快速，持续的得到反馈，并激励新的变化产生（新的商业创新，新的模式等）。

持续交付包含了自动化构建，自动化测试以及自动化部署等过程，持续改进开发流程中的问题，并促进开发人员，测试人员，运维人员之间的协作，团队可以在分钟级别将变更发布上线。

持续交付相关技术及实践

- 版本控制（配置管理）
- 持续集成CI
- 自动化测试
- 构建工具及构建脚本
- 部署流水线

团队通过版本控制来进行协作，所有的代码会在持续集成环境中编译，代码静态检查/分析，自动化测试（还可能产生报告等）。除此之外，CI还还需要有自动化验收测试，自动化回归测试等。

持续交付则更进一步，它将环境准备，持续集成，自动化部署等放在了一起。通过全自动（有些过程可以设置为手动，比如发布到产品环境）的方式，使得软件可以一键发布。如果上线后发现严重defect，还支持一键回滚的机制（其实就是将之前的一个稳定版本做一次发布，由于发布流程已经经过千锤百炼，所以发布本身就变得非常轻松，安全）

这篇文章中，我们会使用 `git` + `github` 作为版本控制工具， `travis-ci` 作为持续集成环境， `gradle` 作为构建工具， `Heroku` 作为应用的部署环境。这些工具都是免费服务，如果你需要更高级的功能（比如更多的并发数，更大的数据库），则可以选择付费套餐。不过对于我们平时的大部分side project来说，免费服务已经足够。

实例

我在《前后端分离了，然后呢？》这篇文章中，提到了一个叫做 `bookmarks` 的应用，这个应用是一个前后端分离的非常彻底的应用。

我们这里会再次使用这个应用作为实例，并采用不同的两个免费服务（travis-ci和snap-ci）来完成 `持续部署` 环境的搭建。

bookmarks服务器

`bookmarks-server` 是一个基于 `spring-boot` 的纯粹的 `API` ，它可以被打包成一个 `jar` 包，然后通过命令行启动运行。在本文中，我们我们将会将这个server部署到heroku平台上。

首先需要定义一个 `Procfile` ，这个是我们应用的入口， `heroku` 根据这个文件来明确以何种方式来启动我们的应用：

```
web: java -Dserver.port=$PORT -jar build/libs/bookmarks-server-0.1.0.jar --spring.profiles.active=staging
```

由于我们在本地使用的使用 `mysql` ，而 `heroku` 默认的是 `postgres` 数据库，因此需要在 `application.yml` 中额外配置

```
spring:
  profiles: staging

  datasource:
    driverClassName: org.postgresql.Driver
    url: ${JDBC_DATABASE_URL}
    username: ${DATABASE_USER}
    password: ${DATABASE_PASS}

  jpa:
    database_platform: org.hibernate.dialect.PostgreSQLDialect
    hibernate:
      ddl-auto: update
```

有了这些配置后，我们需要创建一个 `heroku` 应用：

```
$ heroku create
Created http://quiet-atoll-8237.herokuapp.com/ | git@heroku.com:quiet-atoll-8237.git
```

创建之后，我们可以在界面上对这个应用进行一些配置（当然，也可以通过命令行，具体参看 `heroku help` ）。为了支持数据库，需要为我们的应用添加一个 `postgres` 的AddOn。添加之后， `heroku` 会为我们提供一个 `postgres` 的连接地址，格式大概是这样：

```
postgres://username:password@host:port/database
```

然后我们需要在 `Heroku` 的配置界面中配置一些环境变量：



这样，当应用部署到 `Heroku` 上之后，我们的应用就可以读到这些配置了（注意 `application.yml` 中的环境变量 `JDBC_DATABASE_URL` ）。

搭建持续集成环境

持续集成环境，这里我们选用最简单的 `travis-ci` ，它可以很容易的与 `github` 集成。

- 在项目X中定义一个 `.travis.yml` 的文件
- 将你的代码push到github上
- 绑定github帐号到 `travis`
- 在 `travis` 中启用项目X

这个 `.travis.yml` 因项目而异，我们这里的项目是 `spring-boot` ，所以只需要指定 `java` 即可：

```
language: java
```

如果是 `java` 项目，并且项目中有 `build.gradle` ， `travis-ci` 会自动执行 `gradle check` 任务。

自动化部署

当CI运行成功之后，我们需要 `travis-ci` 帮我们将应用程序发布到 `heroku` 上，这时候需要做一些修改。最简单的方式是直接安装 `travis-ci` 的命令行工具到本地：

```
$ gem install travis -v 1.8.0 --no-rdoc --no-ri
```

然后通过 `heroku` 的 `auth:token` 命令获得 `heroku` 的token，在加密并写入 `.travis.yml` ：

```
$ heroku auth:token
00xxxxxxxxxxxxx55d11dbd0cxxxxxxxxxxxfe067

$ travis encrypt 00xxxxxxxxxxxxx55d11dbd0cxxxxxxxxxxxfe067 --add
```

当然可以合并为一条命令：

```
$ travis encrypt $(heroku auth:token) --add
```

将加密过的token存入 `.travis.yml` 文件。最后的结果大致如下：

```
language: java
deploy:
  provider: heroku
  api_key:
    secure: ...
  app: quiet-atoll-8237
```

注意此处的 `app` ，正是我们的App的名字。另外，还需要给 `build.gradle` 添加一个名叫 `stage` 的task， `travis` 在deploy时需要这个 `task` ：

```
task stage {
    dependsOn build
}
```

```
349  :check
350
351  BUILD SUCCESSFUL
352
353  Total time: 27.098 secs
354
355  The command "./gradlew check" exited with 0.
356  Fetching: dpl-1.8.11.gem (100%)
359
360  Installing deploy dependencies
372  Preparing deploy
379  Deploying application
419  Already up-to-date!
420  # HEAD detached at e26a986
421  nothing to commit, working directory clean
422  Dropped refs/stash@{0} (294e40ef7016cb9a8bdb90c512552fe233438bee)
423
424  Done. Your build exited with 0.
```

这样，我们只需要在本地的一个提交，一切都会自动化起来：

- travis会执行`gradle check`
- `gradle check`会编译并运行自动化测试
- `travis`会部署应用到`heroku`上
- `heroku`会自动重启服务

我们可以在本地进行简单的测试（注意此处我们的`staging`环境的URL）：

```
$ curl https://quiet-atoll-8237.herokuapp.com/api/feeds -s | jq .
[
  {
    "id": 1,
    "url": "http://icodeit.org/2016/01/how-to-summarize-privious-project/",
    "title": "如何持久化你的项目经历",
    "author": "icodit.org",
    "summary": "通常来说，下项目总是一件比较高兴的事（大部分团队还会一起吃个饭庆祝一下）。",
    "publishDate": "2016-01-07"
  },
  {
    "id": 2,
    "url": "http://icodeit.org/2015/11/get-started-with-reflux/",
    "title": "你为什么应该试一试Reflux？",
    "author": "icodit.org",
    "summary": "React在设计之初就只关注在View本身上，其余部分如数据的获取，事件处理等，全然不在考
虑之内。",
    "publishDate": "2016-01-09"
  }
]
```

完整的代码在这里。

其他

CI monitor

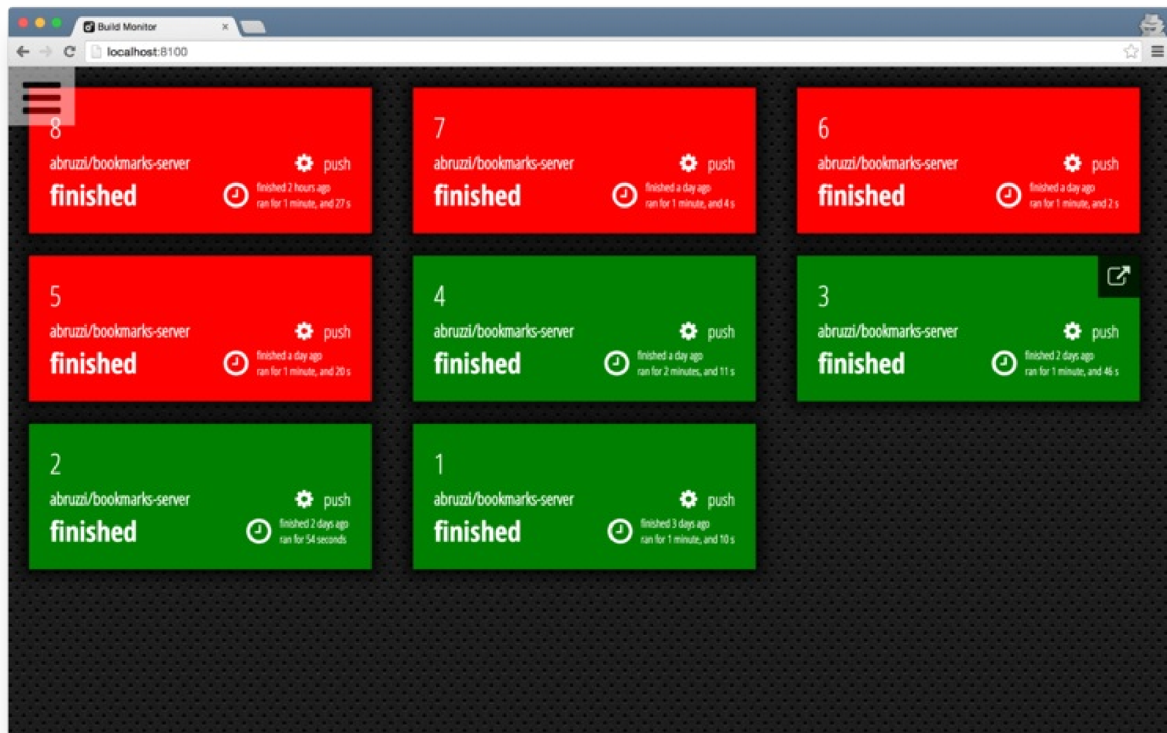
node-build-monitor是一个非常容易配置，使用的CI monitor，我们只需要进行简单地配置，就可以将 `travis` 的状态可视化出来

```
{
  "monitor": {
    "interval": 2000,
    "numberOfBuilds": 12,
    "debug": true
  },
  "services": [
    {
      "name": "Travis",
      "configuration": {
        "slug": "abruzzi/bookmarks-server"
      }
    }
  ]
}
```

不过这个工具会在有网络异常时自动终止，我们可以通过一个简单的脚本来在它终止时自动重启：

```
#!/bin/bash

until node app/app.js
do
    echo "restarting..."
done
```



小结

通过 `travis` 和 `heroku` 这样的免费服务，我们就可以轻松的将自己的项目做到持续集成+持续交付。我们后端的服务相对来说是比较容易的，但是涉及到一个前后端分离的架构，如何做到静态内容的托管，打包，部署，并和后端API集成起来，我会在下一篇文章中详细解释。

客户端程序的的持续交付

上篇文章介绍了如何使用一些免费的服务来实现服务器端API的持续集成、持续交付环境的搭建。有了服务端，自然需要有消费者，在本文中我们将使用另外一个工具来实现纯前端的站点的部署。

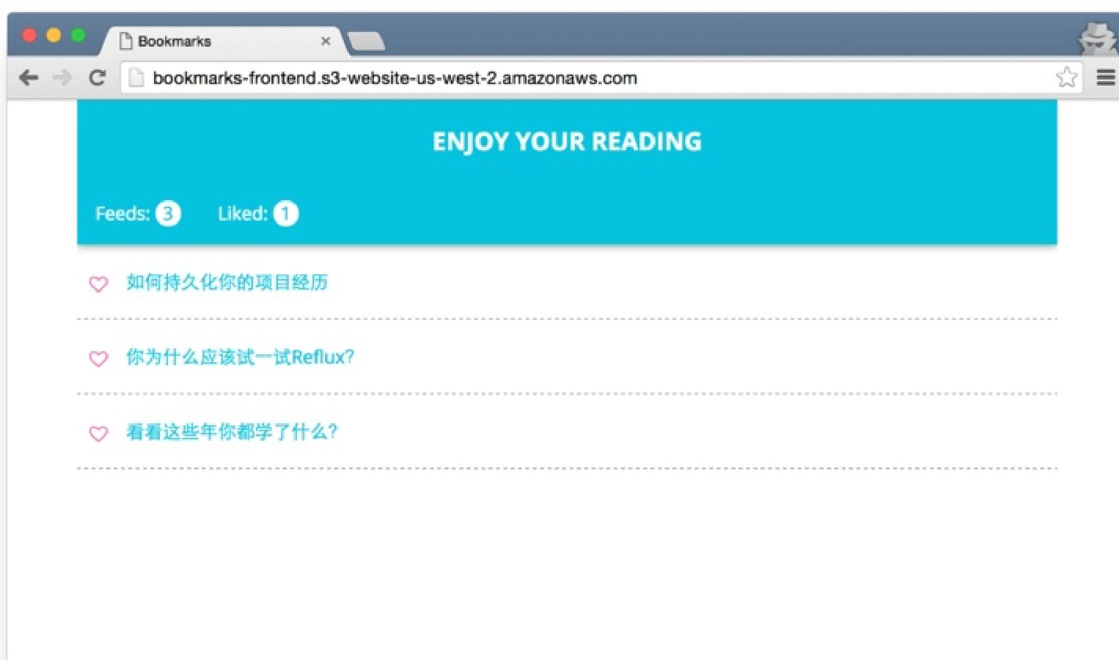
其中包括：

- 持续集成（单元测试，集成测试等）
- 持续部署/持续交付
- 静态站点托管

除此之外，我们还会涉及到：

- 自动化UI测试site_prism
- 静态站点的发布脚本
- aws的命令行工具

我们的应用最后看起来是这样子的。



技术选型

我们在本文中，将采取另外一套免费服务来完成环境的搭建

- ThoughtWorks出品的Snap CI作为持续集成/持续交付环境
- AWS的S3作为应用发布的地方

`Snap CI` 是一个非常易于使用的持续交付环境，由于很多关于持续集成，持续交付的概念和实践都跟 `ThoughtWorks` 有关，所以这个产品对于构建，流水线，部署等等的支持也都做的非常好。

`S3` 是亚马逊的云存储平台，我们可以将静态资源完全托管在其上。 `S3` 的另一个好处是它可以将你的文件变成一个Web Site，比如你的目录中有 `index.html` ，这个文件就可以作为你的站点首页被其他人访问。这个对于我们这个前后端分离项目来说非常有用，我们的 `css` ， `js` ， `font` 文件，还有入口文件 `index.html` 都可以托管于其上。

实例

在本文的例子中，我们将定义3个 `stage` 。 `Snap CI` 将一次发布分为若干个 `stage` ，每个 `stage` 只做一件事情，如果一个 `stage` 失败了，后边的就不会接着执行。

我们的3个 `stage` 分别为：

1. 单元测试
2. 集成测试
3. 部署

准备工作

`bookmarks-frontend` 是一个纯前端的应用，它会消费后端提供的API，但是其实它并不知道（也不应该知道）后端的API部署在什么地方：

```
$(function() {
    var feeds = $.get(config.backend+'/api/feeds');
    var favorite = $.get(config.backend+'/api/fav-feeds/1');

    $.when(feeds, favorite).then(function(feeds, favorite) {
        //...
    });
});
```

由于我们在本地开发时，需要 `backend` 指向本地的服务器，而发布之后，则希望它指向上一篇文章中提到的服务器，因此我们需要编写一点构建脚本来完成这件事儿：

```
var backend = 'http://quiet-atoll-8237.herokuapp.com';

gulp.task('prepareConfig', function() {
    gulp.src(['assets/templates/config.js'])
    .pipe(replace(/#backend#/g, 'http://localhost:8100'))
    .pipe(gulp.dest('assets/script/'));
});

gulp.task('prepareRelease', function() {
    gulp.src(['assets/templates/config.js'])
    .pipe(replace(/#backend#/g, backend))
    .pipe(gulp.dest('assets/script/'));
});
```

我们定义了两个 `gulp` 的task，本地开发时，使用 `prepareConfig` ，要发布时，使用 `prepareRelease` ，然后定义一个简单的模板文件 `config.js` ：

```
module.exports = {
    backend: '#backend#'
}
```

然后可以很简单的包装一下，方便本地开发和发布：

```
gulp.task('dev', ['prepareConfig', 'browserify', 'concatcss']);
gulp.task('build', ['prepareConfig', 'script', 'css']);
gulp.task('release', ['prepareRelease', 'script', 'css']);
```

这样，我们在本地开发时，只需要简单的执行：

```
$ gulp
```

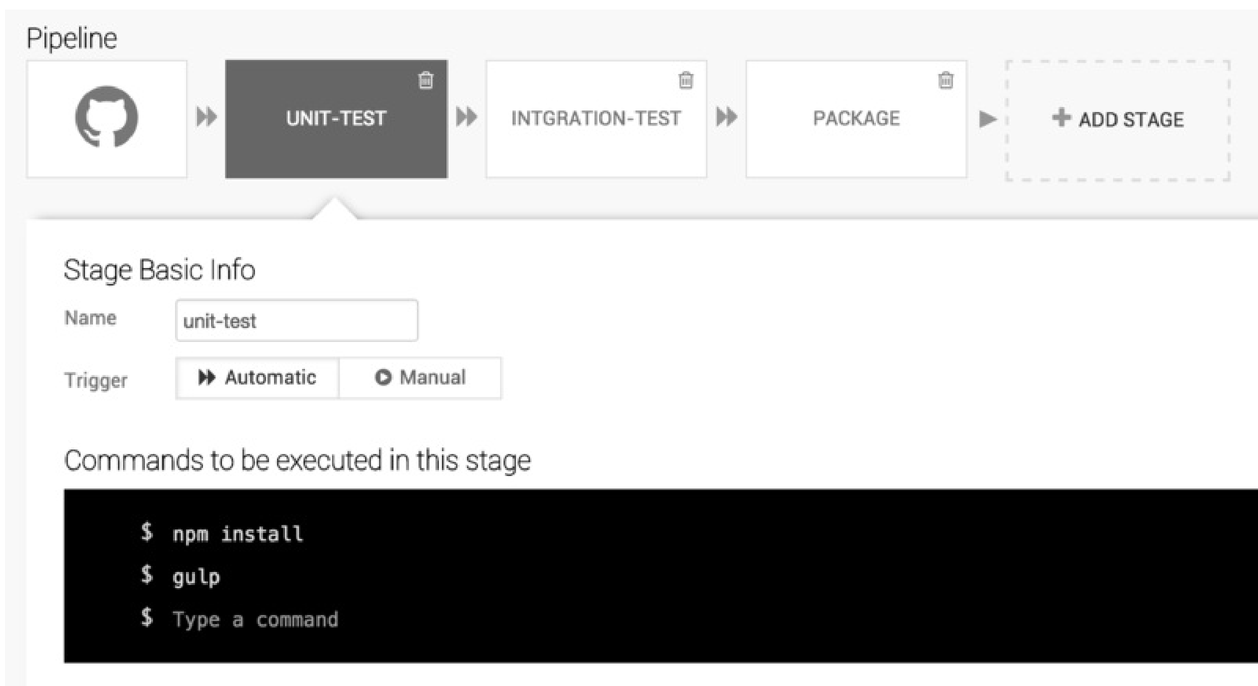
即可。而在发布阶段，只需要执行：

```
$ gulp release
```

单元测试

我们在 `Snap CI` 上将 `github` 上的代码库关联起来，然后添加一个名叫 `unit-test` 的 `stage` ，指定这个 `stage` 对应的命令为：

```
npm install
gulp
```



这样，每当我们有新的提交之后， `Snap CI` 都会拿到新代码，并执行上述命令，如果执行成功，则本地构建成功。

集成测试

由于采取的是前后端分离的策略，我们的应用可以完全独立与后端进行开发，因此我们设置了一个 `fake server` ，具体细节可以参考我之前的博客，也可以看源码。不过这里我们要为集成测试编写一个脚本，并在 `Snap CI` 上执行。

```
#!/bin/bash

export PORT=8100
bundle install

# launch the application
echo "launch the application"
ruby app.rb 2>&1 &
PID=$!

# wait for it to start up
sleep 3

# run the rspec tests and record the status
rspec
RES=$?

# terminate after rspec
echo "terminate the application"
kill -9 $PID

# now we know whether the rspec success or not
exit $RES
```

这个脚本中，首先安装所有的 `gems` ，然后启动 `fake server` 并将这个server放置在后台运行，然后执行 `rspec` 。当 `rspec` 测试执行完成之后，我们终止服务进行，然后返回结果状态码。

这里使用了 `capybara` 和 `poltergeist` 来做UI测试， `capybara` 会驱动 `phantomjs` 来在内存中运行浏览器，并执行定义好的 `UI` 测试，比如此处，我们的UI测试：

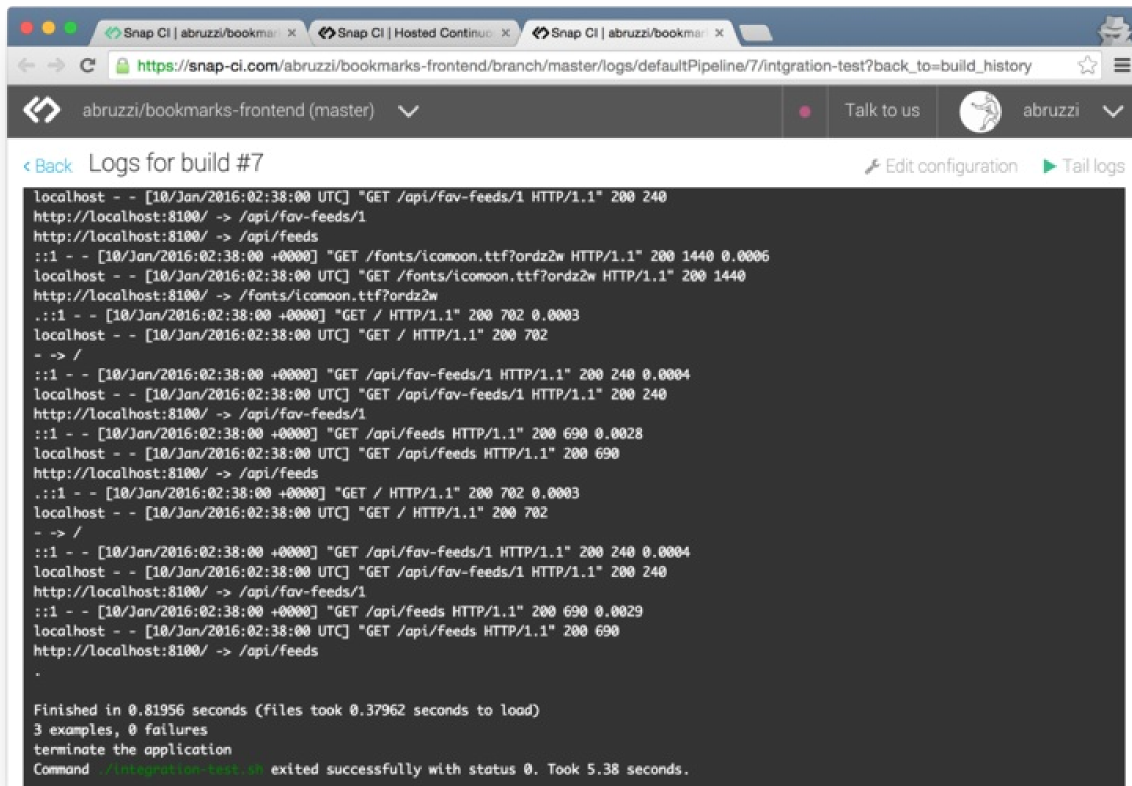
```
require 'spec_helper'

describe 'Feeds List Page' do
    let(:list_page) {FeedListPage.new}

    before do
        list_page.load
    end

    it 'user can see a banner and some feeds' do
        expect(list_page).to have_banner
        expect(list_page).to have_feeds
    end

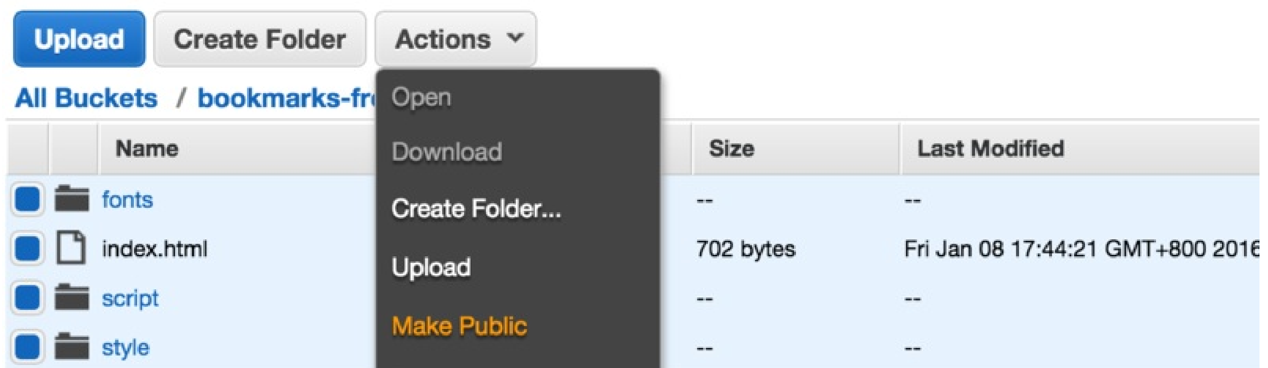
    ##...
end
```



部署

首先需要在 `S3` 上创建一个 `bucket` ，命名为 `bookmarks-frontend` 。然后为其设置 `static website hosting` ，这时候 `AWS` 会assign一个新的域名给你，比如 `http://bookmarks-frontend.s3-website-us-west-2.amazonaws.com/` 。

然后你需要将这个 `bucket` 设置成 `public` ，这样其他人才可以访问你的 `bucket` 。



有了这个之后，我们来编写一个小脚本，这个脚本可以将本地的文件上传至S3。

```
#!/bin/bash

# install gulp and its dependencies
npm install

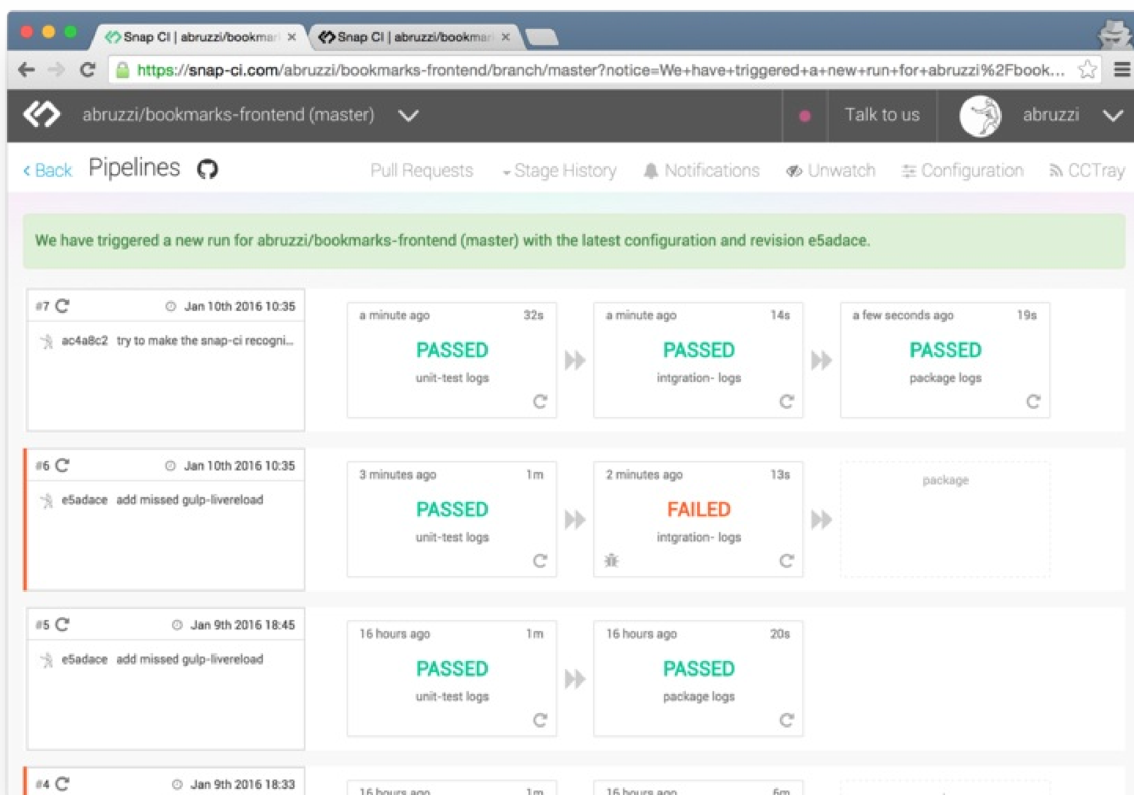
# package stuff, and point the server to the right place
gulp release

# upload the whold folder
aws s3 cp public/ s3://bookmarks-frontend \
    --recursive \
    --region us-west-2 \
    --acl public-read
```

`aws` 命令是 `aws command line` 提供的，另外我们需要在环境变量中设置AWS提供给你的token：

```
AWS_ACCESS_KEY_ID=xxxxxxxxxx
AWS_SECRET_ACCESS_KEY=xxxxxxxxxx
```

然后我们就可以将本地的 `public` 目录递归的上传到S3的对应目录了！



完整的代码可以在此处下载。

总结

我们前端的持续交付也介绍完了。现在前后端应用完全独立，发布也互不影响。不论是服务器端新增加了API，还是添加了新数据，客户端的发布都不受影响；同样，修改样式，添加新的 `JavaScript` 也完全不会影响后端。更重要的是，所有的发布都是一键式的，开发者只需要一个 `git push` 就可以享受这些免费服务提供的自动构建，自动化测试以及自动部署的功能。

Web站点的响应速度

雅虎在2006年就发布了一份提升Web站点响应速度的最佳实践指南。该指南包含了35条规则，分为7个类别。这些规则已经被广泛使用，并指导人们在新的站点设计时更好的考虑问题。

YSlow是一个基于雅虎的这份指南的工具，它可以测试一个站点是否“慢”，以及为什么“慢”？你可以通过很多方式来使用YSlow，比如Firefox，Chrome的插件，命令行工具，甚至PhantomJS这样的无头（Headless）浏览器。

我们这里讨论如何在持续集成服务器上设置一个YSlow任务，这个任务会在每次构建之后，测试你应用的性能指标，以帮助你更快的发现和定位问题。当然，我推荐你在 `staging` 环境，很多开发者在测试环境，本地开发环境都会启动很多对 `Debug` 友好的设置，比如未压缩的JS/CSS，没有超时设置的响应等，这会导致该任务的 `打分` 不够准确。

搭建CI环境

按照传统讨论，我们需要至少做这样一些事情：

- 安装JDK
- 安装Jenkins
- 安装PhantomJS
- 安装YSlow.js脚本

然后设置环境变量，在Jenkins上创建任务，并运行YSlow.js脚本。这个任务很简单，只需要设置好参数，然后将结果输出为Jenkins上的报告即可。比如：

```
$ phantomjs /var/share/yslow.js -i grade -threshold "B" -f junit \
http://bookmarks-frontend.s3-website-us-west-2.amazonaws.com/ > yslow.xml
```

- `-i grade` 展示打分（grade）信息（还可以是basic/stats/all）等
- `-threshold "B"` 指定失败的阈值为B
- `-f junit` 输出为 `junit` 识别的XML格式

这里的阈值可以是数字（0-100分），字母（A-F级别）或者一个JSON字符串（混合使用）

```
-threshold '{"overall": "B", "ycdn": "F", "yexpires": 85}'
```

上面的命令会测试站点 `http://bookmarks-frontend.s3-website-us-west-2.amazonaws.com/` 的各项指标，并应用雅虎的那35条规则，并最终生成一个 `junit` 测试报告格式的文件：`yslow.xml`。

但是维护CI环境是一个比较麻烦的事情，而且既然每个项目都可能会用到这样的`基础设施`，一种好的做法就是将其做成一个`镜像`，这样就可以很容易复用了！这里我们使用docker来安装和配置我们的CI环境，配置完成之后，

基于docker/docker-compose的环境搭建

在`docker`出新之前，我们要搭建一个`测试`或者`staging`环境，需要很多个不同角色的机器：有专门的数据库服务器，文件服务器，缓存服务器，Web服务器，反向代理等等。这样在成本上显然是个不小的开销，如果将所有不同的组件部署在同一台机器上，则又可能互相干扰，只需要一个小小的失误，整个系统就需要重新配置。更可怕的是，这个环境和生产系统不一致，那么很可能真实的错误要等到系统上线之后才会被发现。

2012年，我在一个项目上，系统采用非常传统的J2EE架构。本地开发中，我们使用了Jetty作为容器，而`Staging`使用了Tomcat。Tomcat对空格的处理和Jetty有所不同，我们本地测试通过的代码，在`Staging`完全不能工作。

`docker`出现之后，我们可以在一台物理机器上运行多个互不干涉的容器，每个容器可以是一个组件（比如运行数据库的容器，Web服务器容器等等）。但是对多个容器的管理又是一个很麻烦的事情，`docker`提供了`docker-compose`工具来解决这个问题。使用`docker-compose`可以定义一组互相独立，但是又可以协作在一起的容器，这样我们可以很容易的搭建一个真实的环境。

比如我们可以定义个`docker-compose.yml`

```
app:
  build: .
  links:
    - db:postgres
  ports:
    - 8000:8000
  volumes:
    - .:/app
  working_dir: /app
  entrypoint: /app/start.sh
  environment:
    JDBC_DATABASE_URL: jdbc:postgresql://postgres:5432/bookmarks
    DATABASE_USER: bookmarks-user
    DATABASE_PASS: bookmarks-password

db:
  image: postgres:9.3
  ports:
    - 5432:5432
  environment:
    POSTGRES_DB: bookmarks
    POSTGRES_USER: bookmarks-user
    POSTGRES_PASSWORD: bookmarks-password
```

这个 `docker-compose` 定义了两个组件， `app` 和 `db` 。 `db` 使用了 `postgres:9.3` 镜像，并设置了自己的环境变量。 `app` 则从当前目录 `.` 构建一个新的镜像， `app` 与 `db` 通过 `links` 属性连接起来。

如果在当前目录执行 `docker-compose build` 命令， `docker-compose` 会找到本地的 `Dockerfile` ，然后构建出一个 `docker` 的镜像，并启动该容器，同时，它还会启动 `postgres:9.3` 容器作为数据库组件。这样我们的环境就被完整的搭建好了。

搭建CI环境

```
app:
  build: .
  ports:
    - 8080:8080
    - 50000:50000
  volumes:
    - ./data:/var/jenkins_home
```

这个配置，表明我们会根据当前目录的 `Dockerfile` 来构建一个镜像。

通过命令 `volumns` ，我们将本地目录 `./data` 映射为 `jenkins_home` ，这样我们定义的job信息，以及插件的安装都会放到本地的目录中，方便管理。配置完成之后，构建并启动该容器：

```
FROM jenkins:latest

# Env
ENV PHANTOMJS_VERSION 1.9.6
ENV PHANTOMJS_YSLOW_VERSION 3.1.8
ENV SHARE_BIN /var/share

# Install stuff by using root
USER root
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y git wget libfreetype6 libfontconfig bzip2

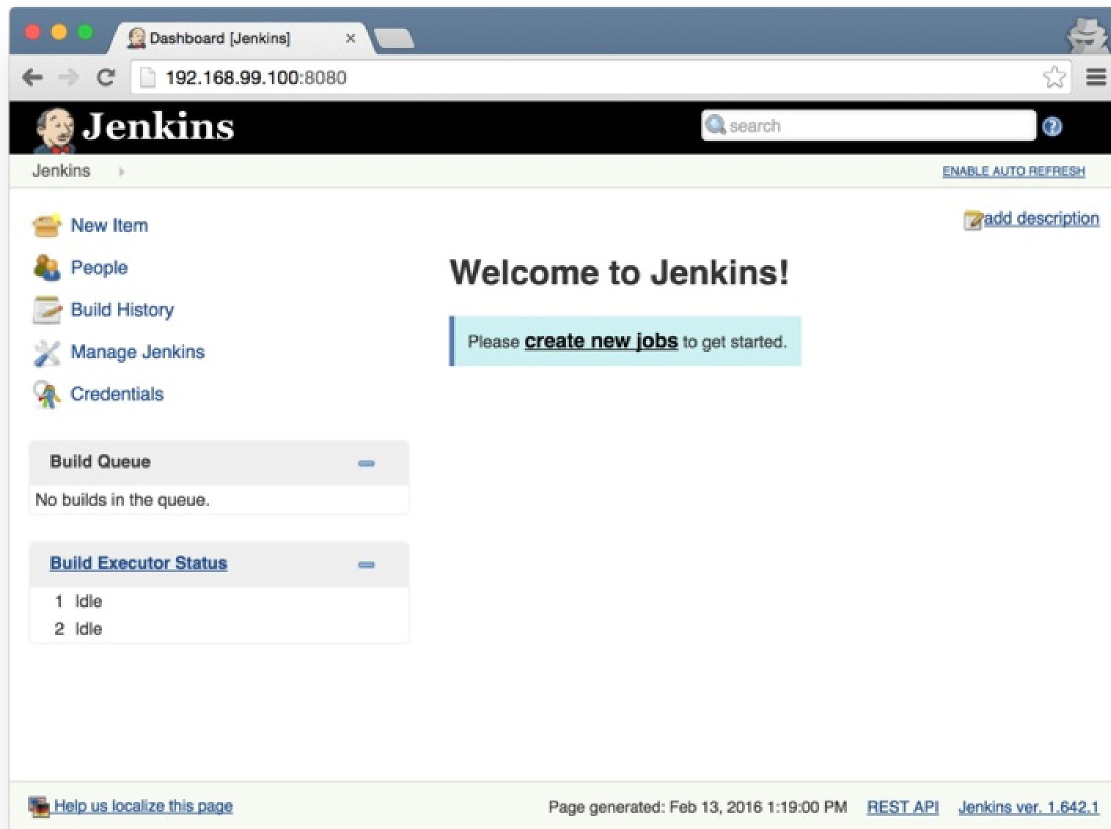
RUN mkdir -p $SHARE_BIN

RUN wget -q --no-check-certificate -O /tmp/phantomjs-$PHANTOMJS_VERSION-linux-x86_64.t
ar.bz2 \
https://bitbucket.org/ariya/phantomjs/downloads/phantomjs-$PHANTOMJS_VERSION-linux-x86
_64.tar.bz2
RUN tar -xjf /tmp/phantomjs-$PHANTOMJS_VERSION-linux-x86_64.tar.bz2 -C /tmp
RUN rm -f /tmp/phantomjs-$PHANTOMJS_VERSION-linux-x86_64.tar.bz2
RUN mv /tmp/phantomjs-$PHANTOMJS_VERSION-linux-x86_64/ $SHARE_BIN/phantomjs
RUN ln -s $SHARE_BIN/phantomjs/bin/phantomjs /usr/bin/phantomjs

RUN wget -q --no-check-certificate -O /tmp/yslow-phantomjs-$PHANTOMJS_YSLOW_VERSION.zi
p \
http://yslow.org/yslow-phantomjs-$PHANTOMJS_YSLOW_VERSION.zip
RUN unzip /tmp/yslow-phantomjs-$PHANTOMJS_YSLOW_VERSION.zip -d $SHARE_BIN/
USER jenkins
```

执行下面的命令来设置并启动CI服务器：

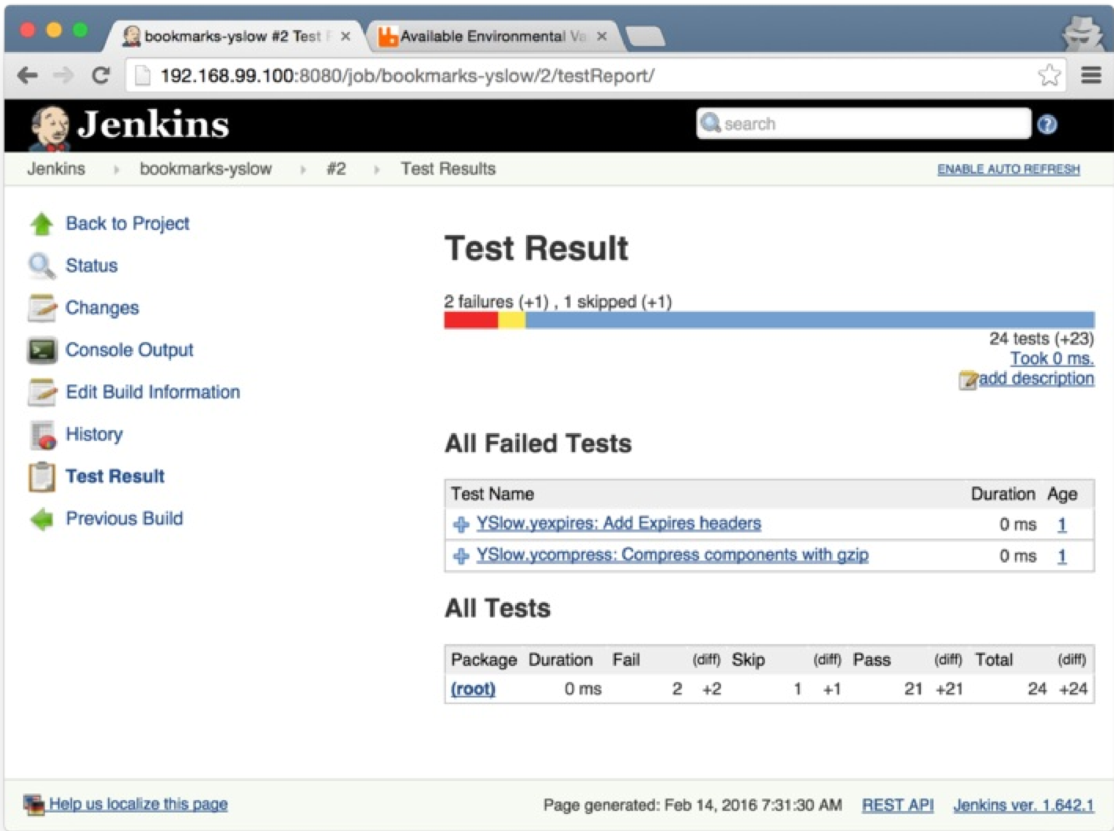
```
docker-compose up
```



创建新任务，并指定该任务执行的命令为：

```
$ phantomjs /var/share/yslow.js -i grade -threshold "B" -f junit \
http://bookmarks-frontend.s3-website-us-west-2.amazonaws.com/ > yslow.xml
```

由于此时 `phantomjs` 已经被安装到了容器中，我们可以直接在jenkins中使用。运行结束之后，这个命令会生成一个报告：



- 没有压缩内容
- 没有添加过期的头信息

在产品环境，我们需要使用反向代理来添加这些头信息，以保证最终用户在使用Web站点时的体验。

配置外化

应用开发中，开发者会将诸如数据库配置信息，NFS服务器的地址，消息队列的大小等等信息保存到配置文件中。比如Java Web中的 `application.properties` 文件，Rails中的 `database.yml` 等。这样我们可以在不同的环境中方便切换，只需要修改几行配置信息，应用的代码则完全不用修改。

下面是一个 `Rails` 应用的数据库配置文件：

```
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: mysql2
  database: test-db
  pool: 5
  username: root
  password: s3cr3t
  socket: /tmp/mysql.sock
```

这个 `yml` 定义了 `test` 环境，数据库使用 `sqlite3` ，数据库文件为 `db/development.sqlite3` 。而在 `production` 环境，数据库采用 `mysql` 。

在运行时，只需要指定环境变量，即可切换数据库：

```
$ RAILS_ENV=test rails server
```

实例

我们以一个简单的Java应用来演示如何将应用程序的配置信息。在这个应用程序中，我们需要数据库配置可以在运行是改变，而不是将配置内置在应用程序中。

在实际场景中，应用程序可能在部署时才知道要连接的数据库地址是什么，而且数据库的名称，数据库连接池的大小等信息都可能因环境而变化。

在这个应用程序中，我们将在应用程序中连接 `mongodb` 数据库，从数据库中读取一个集合的内容，然后打印出整个集合。我们会使用 `soring` 和 `spring-data-mongodb` 来完成简化应用的编写。

我们使用 `gradle` 的 `init` 命令来生成一个典型的Java应用程序：

```
$ mkdir -p spring-mongo-demo
$ cd spring-mongo-demo
$ gradle init --type=java-library
```

另外我们为应用程序添加依赖：

```
apply plugin: 'java'
apply plugin: 'idea'

buildscript {
    repositories {
        jcenter()
    }
}

repositories {
    jcenter()
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.13'
    compile 'org.springframework:spring-context:4.2.4.RELEASE'
    compile 'org.springframework.data:spring-data-mongodb:1.8.4.RELEASE'
}
```

这样，只需要执行

```
$ gradle build
```

就可以下载所有依赖库了。

配置文件

我们首先来为应用程序创建这样的包接口：

```
src
├── main
│   ├── java
│   │   └── com
│   │       └── thoughtworks
│   │           └── mongo
│   │               ├── config
│   │               ├── model
│   │               └── repo
│   └── resources
└── test
    └── java
```

要做到在运行时可改变配置，我们首先需要保证配置和代码分离。这个步骤很容易实现，只需要将配置定义在配置文件中，然后应用在启动时读取该配置（application.properties）即可：

```
mongo.host=localhost
mongo.database=test
```

根据惯例，`application.properties` 会放在 `resources` 目录下。

在我们这个应用中，数据库中又一个 `people` 的集合，对应的我们需要又一个 `Person` 的实体，和一个用来访问数据库集合的 `PersonRepository` 。

```java
package com.thoughtworks.mongo.model;

import org.springframework.data.annotation.Id;

public class Person {
    @Id
    private String id;

    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    //getter & setter

    @Override
    public String toString() {
        return "{name="+name+", age="+age+"}";
    }
}
```

`spring-data-mongo` 提供了一个 `MongoRepository` 接口，我们的应用只需要继承这个接口，就可以免费获得很多有用的数据库访问功能。

```java
package com.thoughtworks.mongo.repo;

import com.thoughtworks.mongo.model.Person;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PersonRepository extends MongoRepository<Person, String> {
}
```

借助 `spring` 强大的注入器，我们很容易在应用中使用这个接口，而不用关心背后它是如何被实例化的：

```
public class Application {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.
class);
        PersonRepository personRepository = context.getBean(PersonRepository.class);

        List<Person> all = personRepository.findAll();
        System.err.println(all);
    }

}
```

我们首先创建一个基于注解的Context，具体的应用配置我们放在了 `AppConfig` 类中，有了这个Context，我们可以从中获取 `PersonRepository` 的实例，并使用它的 `findAll` 方法来获取所有的人员列表。

对于我们的应用来说，所有的配置信息都放在 `AppConfig` 中。我们来看看这个类：

```
package com.thoughtworks.mongo.config;

import com.mongodb.MongoClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.*;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;
import org.springframework.data.mongodb.MongoDbFactory;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.SimpleMongoDbFactory;

@Configuration
@ComponentScan(value = "com.thoughtworks.mongo.*")
@PropertySource(value = "classpath:application.properties")
public class AppConfig {
    @Autowired
    private Environment environment;

    @Bean
    public MongoDbFactory mongoDbFactory() throws Exception {
        String mongoHost = environment.getProperty("mongo.host");
        String mongoDatabase = environment.getProperty("mongo.database");

        return new SimpleMongoDbFactory(new MongoClient(mongoHost), mongoDatabase);
    }

    @Bean
    public MongoTemplate mongoTemplate() throws Exception {
        return new MongoTemplate(mongoDbFactory());
    }

    @Bean
    public static PropertySourcesPlaceholderConfigurer propertyConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

}
```

这个类中，我们使用了 `PropertySource` 这个注解，并指定了配置文件应该从 `classpath` 中的 `application.properties` 中获取。

当我们执行应用时，配置文件会生效，一起正常！下来我们来构建一个可以独立发布的 `jar` 包，这样任何人都可以直接使用这个jar包，而不需要自己重新下载依赖，重新构建等，所以我们为 `build.gradle` 添加几条简单的命令：

```
jar {
    baseName = rootProject.name
    version =  '0.1.0'

    from {
        configurations.compile.collect { it.isDirectory() ? it : zipTree(it) }
    }

    manifest {
        attributes("Main-Class": "com.thoughtworks.mongo.Application")
    }
}
```

这样构建出来的包就会包含所有依赖，我们还显式的指定了该jar包里的 `Main-Class` 是 `com.thoughtworks.mongo.Application` 。

```
$ gradle build
$ java -jar build/libs/spring-mongo-demo-0.1.0.jar
```

会得到

```
[{name=Juntao, age=30}, {name=Abruzzi, age=28}]
```

很好，我们成功的访问了数据库，并打印出了集合中的所有元素，而且这个应用程序是可以独立发布的了。

系统配置文件

如果仔细想想使用场景，你会发现如果数据库连接发生变化了，我们修改 `application.properties` 文件，而该文件是打包在jar包内的！

这意味这应用程序和它的环境并没有分离，简单来说，我们需要提供机制来让外部的配置可以覆盖包内的配置。

最简单的方式下，我们创建一个新的 `properties` 文件，并让应用程序最后使用这个配置，这样就可以达到覆盖的目的了。当然，如果外部没有提供 `properties` 文件，应用还可以使用内部的配置提供功能。

`spring` 在版本4之后，提供了比 `PropertySource` 更强大的注解 `PropertySources` ，它支持定义多个配置源，并形成一个链表，这样后边的元素就可以覆盖前面的元素了。

```
@Configuration
@ComponentScan(value = "com.thoughtworks.mongo.*")
@PropertySources({
        @PropertySource(value = "classpath:application.properties"),
        @PropertySource(value = "file:/etc/spring-mongo/application.properties", ignor
eResourceNotFound=true)
})
public class AppConfig {
    //...
}
```

我们定义了两个配置源，一个是 `application.properties` ，另一个是绝对路径下 `/etc/spring-demo/` 下的同名文件。 `ignoreResourceNotFound=true` 保证如果找不到该配置，也不会报错。

这样，如果我们需要新的数据库连接/配置，只需要在 `/etc/spring-demo` 下创建同名文件，并设置新的值即可。

这样做的好处是，应用程序无需做任何修改，配置信息外化到了环境中，部署应用程序的环境来确定应用具体如何与外部依赖交互。

环境变量

另一种常用的方式是使用环境变量，这种方式下，我们只需要修改启动脚本，就可以将信息传递给应用程序，这中方式在 `UNIX` 世界已经存在多年。

`spring` 提供了 `Environment` 对象，该对象提供了对环境的封装，其中包含了环境应用了那些 `profile` 的，系统配置，操作系统环境变量，以及所有的配置源 `propertySources` ：

```
{activeProfiles=[], defaultProfiles=[default], propertySources=[systemProperties,syste
mEnvironment,URL [file:/etc/spring-mongo/application.properties],class path resource [
application.properties]]}
```

这样，我们的代码中使用的

```
@Autowired
private Environment environment;

@Bean
public MongoDbFactory mongoDbFactory() throws Exception {
    String mongoHost = environment.getProperty("mongo.host");
    String mongoDatabase = environment.getProperty("mongo.database");

    return new SimpleMongoDbFactory(new MongoClient(mongoHost), mongoDatabase);
}
```

都自然的可以从Java环境变量中获得配置信息：

```
java -Dmongo.host=10.29.10.212 -Dmongo.database=prod-db -jar build/libs/spring-mongo-d
emo-0.1.0.jar
```

这样，配置信息就通过外部传入。这里仅仅是一个很简单的例子，项目中的配置信息会非常多，而且可能会有多种方式混用的场景：部分信息放在系统的环境变量中，如 `/etc/profile` 或者 `.bashrc` 中，另外一部分信息则存储在应用特定的properties中。

总体而言，这些配置信息都需要和应用程序本身分离。这和持续交付的实践其实也是相关的，在持续交付中，我们只会生成一个二进制包。这个二进制包在部署流水线上一直使用，在回归测试、性能测试等任务中，这个包会被部署在不同的环境中，并被测试有效性。这样我们才对应用程序自身有更多的信心。

工程实践

在ThoughtWorks，几乎每个团队都在遵循着某些敏捷实践，但是每个团队的实践都有不同。敏捷本身的意义就是拥抱变化，持续改进。我怀疑，即使最开始所有团队都学习了同样的实践，经过几个迭代之后，每个团队都会演化出适合自己团队`节奏`的实践集。

我对一些同事做过采访，每个被采访者只需要回答一个问题：

> 假设需要你启动一个项目，项目成员已经就绪，唯一的问题是，客户对敏捷实践抱有怀疑（他已经被市面上的各种`敏捷`吓坏了），因此他不太支持那些`新鲜的`，`反直觉`的敏捷实践。你推行的所有实践都可能会遇到挑战，另外由于有交付压力，你不得不做出一些妥协。 那么，你接受的最低限度的敏捷实践的集合是什么？（也就是说，如果客户和你无法在这个集合上达成一致的话，那么你就会退出该项目）

整理后的一个列表是这样的：

- 可视化的需求卡片墙
- 迭代方式管理
- 持续集成（编译，检查，测试，打包）
- 测试覆盖全面
- 每日站会
- 回顾会议
- 知识传递（结对编程，Git Pull Request，Code Review）
- 版本控制
- 自动化脚本（构建，部署）

版本控制系统

我在2012年的时候，还听说有客户使用基于ftp的源码控制，也就是说本地开发完成之后，通过ftp拷贝到远程服务器并覆盖的方式。这个现象可能现在已经不存在了，我参加过的大部分项目都采用了某种形式的版本控制，很多比较传统的企业会采用`subversion`，而大多数较新的项目都采用了`git`作为版本控制系统。

版本控制系统不但可以帮你将改动痕迹保存起来，还可以很方便的提供`diff`功能，这样你可以和其他人讨论修改内容（后边的Code Review小节我们会详细讨论）。另外，版本控制系统（如git）可以让你回退到历史的某个版本，这样你可以很容易的构建出当时的包并进行测试，这在查找到底是哪次提交引入了缺陷时非常重要（git提供`bisect`字命令，可以更加迅速的定位缺陷引入时的版本）。

无论如何，版本控制系统已经成为开发的标配，我们推荐你使用功能强大的`git`，它提供的丰富特性更适合分布式团队的协作。

迭代开发

闭门造车的时代已经一去不复返了。需求分析3个月，开发半年，测试半年的瀑布方式也与飞速发展的时代不匹配了。快速，轻量级的开发，发布，测试已经成为常态，没有人愿意为软件的某个功能而等上好几个月。

持续集成

自动化测试

站会

回顾会议

知识传递

在 `ThoughtWorks` ，我们有非常多的Session。所谓Session，就是一个小的演讲，我们主题没有限制。比如有人讲Scala中的函数式编程，有人讲业界的自动化测试成熟度，有人讲构建分布式应用的12条原则等等。更多时候，在一个团队内部，通常会有不定时的组内分享，比如有人在过去的两个迭代中使用了某种CSS的预处理器，有人在Linux服务器上自定义了一些服务等等，都可以做为分享的主题。

事实上，为了保证知识在团队内部的高效传递，我们引入了各种各样的实践（除了写文档，因为那东西肯定没人看）。Code Review，结对编程，讲Session都是知识传递很好的方式。当然，如果说没有任何文档也是不对的，每个项目都会有一些wiki页面，团队成员会不定时的更新。

有些知识，或者说信息都会放在wiki上，比如如何在本地搭建Production-like的环境，staging环境的负载均衡器部署在哪一台服务器等等，这些信息都会有记录。不过另一方面，我们尽量将知识内化在团队成员的大脑里，我是说，所有人的大脑里。

自动化一切

测试策略

在软件公司里，人们喜欢强调测试，强调质量。但是如果你在一个传统的软件项目上呆过一段时间的话，就会知道，人们只是喜欢强调而已。人们其实并不知道如何去关心软件质量。

当你问一个项目经理，为了赶进度，你愿意舍弃哪些东西？几乎毫无意外的，项目经理会选择舍弃自动化测试。他们会说，我们需要在deadline之前让领导看到结果！为了赶进度而造出来的软件，脆弱，不堪一击，在生产环境中会为公司带来损失，但是这些都是“系统中另一部分”的事情了。

~~有些公司会为开发人员配置对应的测试人员，开发部门和测试部门的人数几乎能达到1:1。一般而言，开发人员在开发新的版本，测试人员在测试上一个版本。开发人员和测试人员的输入都是`软件设计`部门提供的文档。~~

我的第一家公司里，测试人员和开发人员的比例基本上是1:1。开发在交付一个版本到开始下一个版本的开发之间，会有一个所谓`bug fixing`的阶段，大约4周到6周，专门等待测试提单，然后从一个bug跟踪系统中获取`bug`，然后修复。

开发在修复完成之后，修改bug状态，测试人员会得到通知，然后再测一次，这个过程可能会往复多次。开发没有测试所拥有的完整环境（我们的软件需要安装到Redhat Linux上，而开发机器都是Windows，并且不允许安装虚拟机），所以每次修复之后保证本地可以运行。

整个过程都是手工的，测试人员好像是有一些简单的UI测试脚本，不过由于我们的界面经常变化，她们需要重新录制脚本，过程非常麻烦。

我的第二家公司则是另外一个极端，我们10个开发人员，对应的只有1名QA工程师，每到我们需要给客户showcase的时候，所有人都手忙脚乱的帮助QA做测试，相信你可以想象那种人心惶惶的场景。

在日常的工作中，如果肯花时间来为自己的代码编写自动化测试（单元测试，集成测试，UI测试），无疑人们会对自己的软件有更大的信心。而且事实上软件质量是需要整个团队来保证的，不要把希望寄托在只做黑盒测试的QA身上。

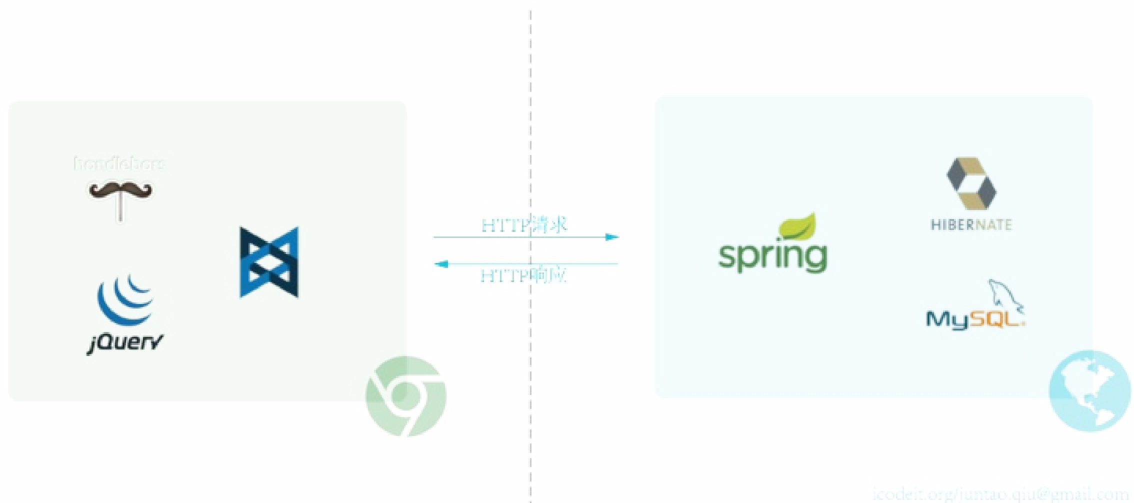
前后端分离

前后端分离已经是业界所共识的一种开发/部署模式了。所谓的前后端分离，并不是传统行业中的按部门划分，一部分人纯做前端（HTML/CSS/JavaScript/Flex），另一部分人纯做后端，因为这种方式是不工作的：比如很多团队采取了后端的模板技术（JSP, FreeMarker, ERB等等），前端的开发和调试需要一个后台Web容器的支持，从而无法做到真正的分离（更不用提在部署的时候，由于动态内容和静态内容混在一起，当设计动态静态分流的时候，处理起来非常麻烦）。关于前后端开发的另一个讨论可以参考这里。

即使通过API来解耦前端和后端开发过程，前后端通过 `RESTful` 的接口来通信，前端的静态内容和后端的动态计算分别开发，分别部署，集成仍然是一个绕不开的问题 --- 前端/后端的应用都可以独立的运行，但是集成起来却不工作。我们需要花费大量的精力来调试，直到上线前仍然没有人有信心所有的接口都是工作的。

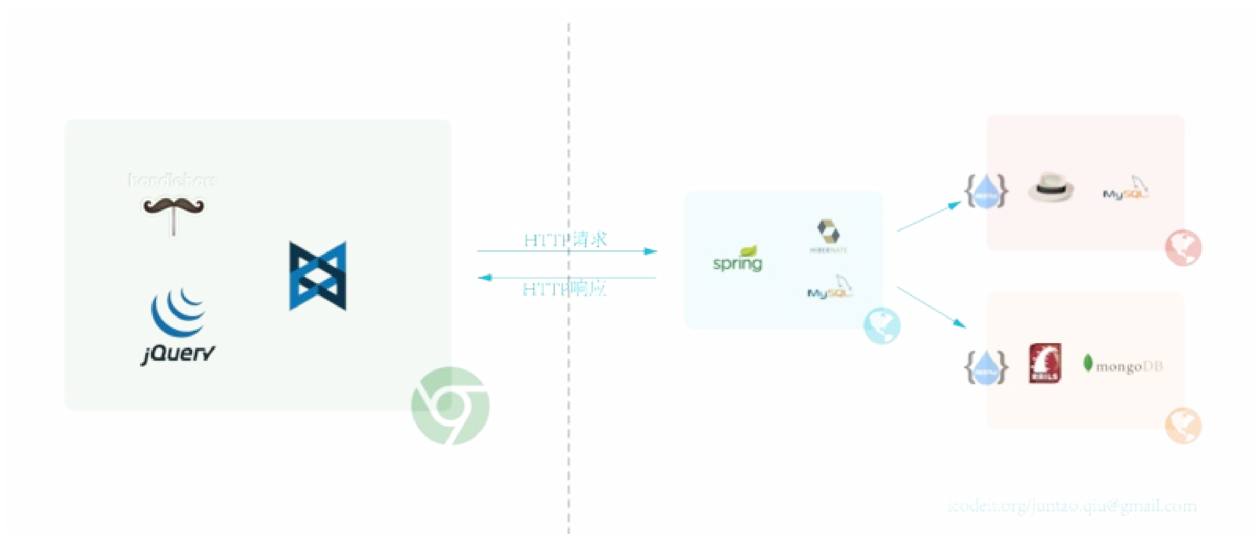
一点背景

一个典型的Web应用的布局看起来是这样的：

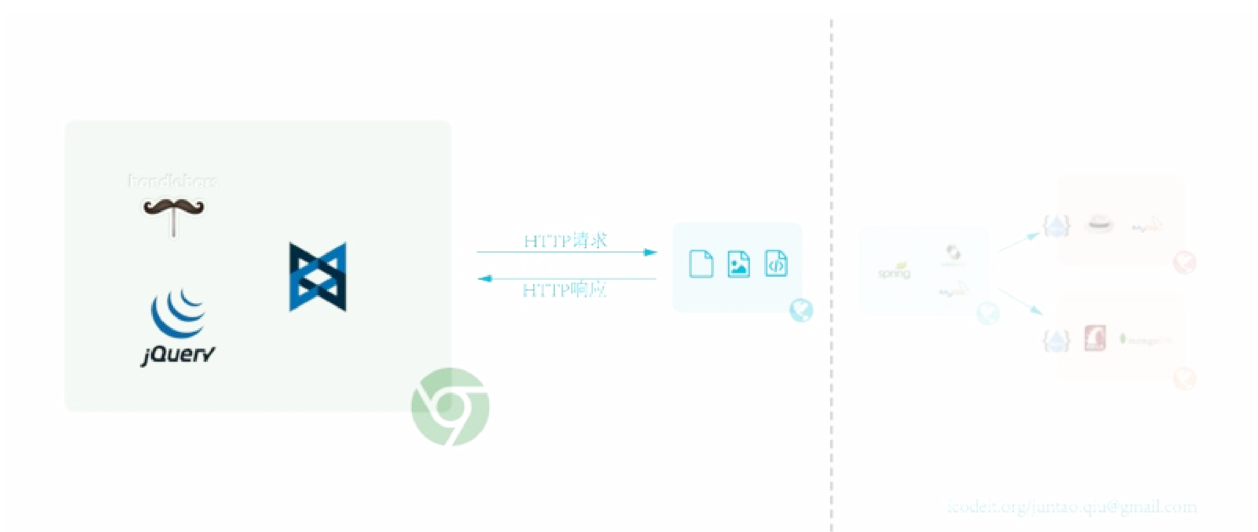


前后端都各自有自己的开发流程，构建工具，测试集合等等。前后端仅仅通过接口来编程，这个接口可能是JSON格式的RESTFul的接口，也可能是XML的，重点是后台只负责数据的提供和计算，而完全不处理展现。而前端则负责拿到数据，组织数据并展现的工作。这样结构清晰，关注点分离，前后端会变得相对独立并松耦合。

上述的场景还是比较理想，我们事实上在实际环境中会有非常复杂的场景，比如异构的网络，异构的操作系统等等:



在实际的场景中，后端可能还会更复杂，比如用C语言做数据采集，然后通过Java整合到一个数据仓库，然后该数据仓库又有一层Web Service，最后若干个这样的Web Service又被一个Ruby的聚合Service整合在一起返回给前端。在这样一个复杂的系统中，后台任意端点的失败都可能阻塞前端的开发流程，因此我们会采用mock的方式来解决这个问题：



这个 `mock` 服务器可以启动一个简单的HTTP服务器，然后将一些静态的内容serve出来，以供前端代码使用。这样的好处很多:

1. 前后端开发相对独立
2. 后端的进度不会影响前端开发
3. 启动速度更快
4. 前后端都可以使用自己熟悉的技术栈（让前端的学maven，让后端的用gulp都会很不顺手）

但是当集成依然是一个令人头疼的难题。我们往往在集成的时候才发现，本来协商的数据结构变了： `deliveryAddress` 字段本来是一个字符串，现在变成数组了（业务发生了变更，系统现在可以支持多个快递地址）； `price` 字段变成字符串，协商的时候是 `number` ；用户邮箱地

址多了一个层级等等。这些变动在所难免，而且时有发生，这会花费大量的调试时间和集成时间，更别提修改之后的回归测试了。

所以仅仅使用一个静态服务器，然后提供 `mock` 数据是远远不够的。我们需要的 `mock` 应该还能做到：

1. 前端依赖指定格式的mock数据来进行UI开发
2. 前端的开发和测试都基于这些mock数据
3. 后端产生指定格式的mock数据
4. 后端需要测试来确保生成的mock数据正是前端需要的

简而言之，我们需要商定一些契约，并将这些契约作为可以被测试的中间格式。然后前后端都需要有测试来使用这些契约。一旦契约发生变化，则另一方的测试会失败，这样就会驱动双方协商，并降低集成时的浪费。

一个实际的场景是：前端发现已有的某个契约中，缺少了一个 `address` 的字段，于是就在契约中添加了该字段。然后在UI上将这个字段正确的展现了（当然还设置了字体，字号，颜色等等）。但是后台生成该契约的服务并没有感知到这一变化，当运行生成契约部分测试（后台）时，测试会失败了 --- 因为它并没有生成这个字段。于是后端工程师就找前端来商量，了解业务逻辑之后，他会修改代码，并保证测试通过。这样，当集成的时候，就不会出现UI上少了一个字段，但是谁也不知道是前端问题，后端问题，还是数据库问题等。

而且实际的项目中，往往都是多个页面，多个API，多个版本，多个团队同时进行开发，这样的契约会降低非常多的调试时间，使得集成相对平滑。

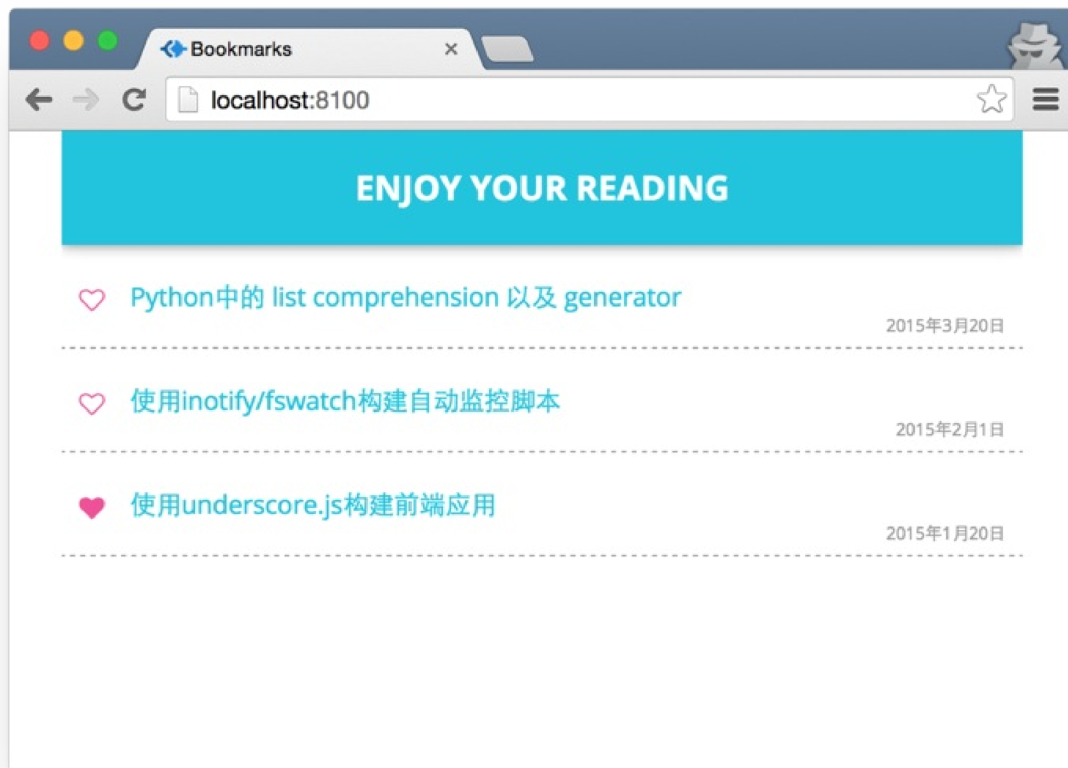
在实践中，契约可以定义为一个JSON文件，或者一个XML的payload。只需要保证前后端共享同一个契约集合来做测试，那么集成工作就会从中受益。一个最简单的形式是：提供一些静态的 `mock` 文件，而前端所有发往后台的请求都被某种机制拦截，并转换成对该静态资源的请求。

1. moco，基于Java
2. wiremock，基于Java
3. sinatra，基于Ruby

看到 `sinatra` 被列在这里，可能熟悉 `Ruby` 的人会反对：它可是一个 `后端` 全功能的的程序库啊。之所以列它在这里，是因为 `sinatra` 提供了一套简洁优美的 `DSL` ，这个 `DSL` 非常契合 `Web` 语言，我找不到更漂亮的方式来使得这个 `mock server` 更加易读，所以就采用了它。

一个例子

我们以这个应用为示例，来说明如何在前后端分离之后，保证代码的质量，并降低集成的成本。这个应用场景很简单：所有人都可以看到一个条目列表，每个登陆用户都可以选择自己喜欢的条目，并为之加星。加星之后的条目会保存到用户自己的 `个人中心` 中。用户界面看起来是这样的：



不过为了专注在我们的中心上，我去掉了诸如登陆，个人中心之类的页面，假设你是一个已登录用户，然后我们来看看如何编写测试。

前端开发

根据通常的做法，前后端分离之后，我们很容易 `mock` 一些数据来自己测试：

```
[
    {
        "id": 1,
        "url": "http://abruzzi.github.com/2015/03/list-comprehension-in-python/",
        "title": "Python中的 list comprehension 以及 generator",
        "publicDate": "2015年3月20日"
    },
    {
        "id": 2,
        "url": "http://abruzzi.github.com/2015/03/build-monitor-script-based-on-inotif
y/",
        "title": "使用inotify/fswatch构建自动监控脚本",
        "publicDate": "2015年2月1日"
    },
    {
        "id": 3,
        "url": "http://abruzzi.github.com/2015/02/build-sample-application-by-using-un
derscore-and-jquery/",
        "title": "使用underscore.js构建前端应用",
        "publicDate": "2015年1月20日"
    }
]
```

然后，一个可能的方式是通过请求这个json来测试前台：

```
$(function() {
    $.get('/mocks/feeds.json').then(function(feeds) {
        var feedList = new Backbone.Collection(extended);
        var feedListView = new FeedListView(feedList);

        $('.container').append(feedListView.render());
    });
});
```

这样当然是可以工作的，但是这里发送请求的 `url` 并不是最终的，当集成的时候我们又需要修改为真实的 `url` 。一个简单的做法是使用 `Sinatra` 来做一次 `url` 的转换：

```
get '/api/feeds' do
    content_type 'application/json'
    File.open('mocks/feeds.json').read
end
```

这样，当我们和实际的服务进行集成时，只需要连接到那个服务器就可以了。

注意，我们现在的核心是 `mocks/feeds.json` 这个文件。这个文件现在的角色就是一个契约，至少对于前端来说是这样的。紧接着，我们的应用需要渲染 `加星` 的功能，这就需要另外一个契约：找出当前用户加星过的所有条目，因此我们加入了一个新的契约：

```
[
    {
        "id": 3,
        "url": "http://abruzzi.github.com/2015/02/build-sample-application-by-using-underscore-and-jquery/",
        "title": "使用underscore.js构建前端应用",
        "publicDate": "2015年1月20日"
    }
]
```

然后在 `sinatra` 中加入一个新的映射：

```
get '/api/fav-feeds/:id' do
    content_type 'application/json'
    File.open('mocks/fav-feeds.json').read
end
```

通过这两个请求，我们会得到两个列表，然后根据这两个列表的交集来绘制出所有的星号的状态（有的是空心，有的是实心）：

```
$.when(feeds, favorite).then(function(feeds, favorite) {
    var ids = _.pluck(favorite[0], 'id');
    var extended = _.map(feeds[0], function(feed) {
        return _.extend(feed, {status: _.includes(ids, feed.id)});
    });

    var feedList = new Backbone.Collection(extended);
    var feedListView = new FeedListView(feedList);

    $('.container').append(feedListView.render());
});
```

剩下的一个问题是当点击红心时，我们需要发请求给后端，然后更新红心的状态：

```
toggleFavorite: function(event) {
    event.preventDefault();
    var that = this;
    $.post('/api/feeds/'+this.model.get('id')).done(function(){
        var status = that.model.get('status');
        that.model.set('status', !status);
    });
}
```

这里又多出来一个请求，不过使用Sinatra我们还是可以很容易的支持它：

```
post '/api/feeds/:id' do
end
```

可以看到，在没有后端的情况下，我们一切都进展顺利 --- 后端甚至还没有开始做，或者正在由一个进度比我们慢的团队在开发，不过无所谓，他们不会影响我们的。

不仅如此，当我们写完前端的代码之后，可以做一个 `End2End` 的测试。由于使用了mock数据，免去了数据库和网络的耗时，这个 `End2End` 的测试会运行的非常快，并且它确实起到了端到端的作用。这些测试在最后的集成时，还可以用来当UI测试来运行。所谓一举多得。

```
#encoding: utf-8
require 'spec_helper'

describe 'Feeds List Page' do
    let(:list_page) {FeedListPage.new}

    before do
        list_page.load
    end

    it 'user can see a banner and some feeds' do
        expect(list_page).to have_banner
        expect(list_page).to have_feeds
    end

    it 'user can see 3 feeds in the list' do
        expect(list_page.all_feeds).to have_feed_items count: 3
    end

    it 'feed has some detail information' do
        first = list_page.all_feeds.feed_items.first
        expect(first.title).to eql("Python中的 list comprehension 以及 generator")
    end
end
```

```
➜  bookmarks-frontend git:(master) rspec -fd

Feeds List Page
  user can see a banner and some feeds
  user can see 3 feeds in the list
  feed has some detail information

Finished in 2.82 seconds (files took 0.97239 seconds to load)
3 examples, 0 failures
➜  bookmarks-frontend git:(master)
```

关于如何编写这样的测试，可以参考之前写的这篇文章。

后端开发

我在这个示例中，后端采用了 `spring-boot` 作为示例，你应该可以很容易将类似的思路应用到Ruby或者其他语言上。

首先是请求的入口，`FeedsController` 会负责解析请求路径，查数据库，最后返回JSON格式的数据。

```
@Controller
@RequestMapping("/api")
public class FeedsController {

    @Autowired
    private FeedsService feedsService;

    @Autowired
    private UserService userService;

    public void setFeedsService(FeedsService feedsService) {
        this.feedsService = feedsService;
    }

    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    @RequestMapping(value="/feeds", method = RequestMethod.GET)
    @ResponseBody
    public Iterable<Feed> allFeeds() {
        return feedsService.allFeeds();
    }


    @RequestMapping(value="/fav-feeds/{userId}", method = RequestMethod.GET)
    @ResponseBody
    public Iterable<Feed> favFeeds(@PathVariable("userId") Long userId) {
        return userService.favoriteFeeds(userId);
    }
}
```

具体查询的细节我们就不做讨论了，感兴趣的可以在文章结尾处找到代码库的链接。那么有了这个Controller之后，我们如何测试它呢？或者说，如何让契约变得实际可用呢？

`spring-test` 提供了非常优美的DSL来编写测试，我们仅需要一点代码就可以将契约用起来，并实际的监督接口的修改：

```
private MockMvc mockMvc;
private FeedsService feedsService;
private UserService userService;

@Before
public void setup() {
    feedsService = mock(FeedsService.class);
    userService = mock(UserService.class);

    FeedsController feedsController = new FeedsController();
    feedsController.setFeedsService(feedsService);
    feedsController.setUserService(userService);

    mockMvc = standaloneSetup(feedsController).build();
}
```

建立了mockmvc之后，我们就可以编写Controller的单元测试了：

```
@Test
public void shouldResponseWithAllFeeds() throws Exception {
    when(feedsService.allFeeds()).thenReturn(Arrays.asList(prepareFeeds()));

    mockMvc.perform(get("/api/feeds"))
            .andExpect(status().isOk())
            .andExpect(content().contentType("application/json;charset=UTF-8"))
            .andExpect(jsonPath("$", hasSize(3)))
            .andExpect(jsonPath("$[0].publishDate", is(notNullValue())));
}
```

当发送 `GET` 请求到 `/api/feeds` 上之后，我们期望返回状态是200，然后内容是 `application/json` 。然后我们预期返回的结果是一个长度为3的数组，然后数组中的第一个元素的 `publishDate` 字段不为空。

注意此处的 `prepareFeeds` 方法，事实上它会去加载 `mocks/feeds.json` 文件 --- 也就是前端用来测试的mock文件：

```
private Feed[] prepareFeeds() throws IOException {
    URL resource = getClass().getResource("/mocks/feeds.json");
    ObjectMapper mapper = new ObjectMapper();
    return mapper.readValue(resource, Feed[].class);
}
```

这样，当后端修改 `Feed` 定义（添加/删除/修改字段），或者修改了mock数据等，都会导致测试失败；而前端修改mock之后，也会导致测试失败 --- 不要惧怕失败 --- 这样的失败会促进一次协商，并驱动出最终的service的契约。

对应的，测试 `/api/fav-feeds/{userId}` 的方式类似：

```
@Test
public void shouldResponseWithUsersFavoriteFeeds() throws Exception {
    when(userService.favoriteFeeds(any(Long.class)))
        .thenReturn(Arrays.asList(prepareFavoriteFeeds()));

    mockMvc.perform(get("/api/fav-feeds/1"))
            .andExpect(status().isOk())
            .andExpect(content().contentType("application/json;charset=UTF-8"))
            .andExpect(jsonPath("$", hasSize(1)))
            .andExpect(jsonPath("$[0].title", is("使用underscore.js构建前端应用")))
            .andExpect(jsonPath("$[0].publishDate", is(notNullValue())));
}
```

总结

前后端分离是一件容易的事情，而且团队可能在短期可以看到很多好处，但是如果不认真处理集成的问题，分离反而可能会带来更长的集成时间。通过面向契约的方式来组织各自的测试，可以带来很多的好处：更快速的 `End2End` 测试，更平滑的集成，更安全的分离开发等等。

代码

前后端的代码我都放到了Gitbub上，感兴趣的可以clone下来自行研究：

1. bookmarks-frontend
2. bookmarks-server

附录

Code Review

在 `ThoughtWorks` ，我们几乎每天都会进行一个叫 `code review` 或者 `code diff` 的活动：每天下午5:00，团队成员围坐在一起，将今天的修改大概过一下，这样做的好处非常明显：

1. 分享业务知识，了解彼此的工作
2. 分享技术细节，比如有人使用了某种设计模式
3. 帮助别人发现问题，比如逻辑错误等，群策群力



经过实践， `code reivew` 可以快速发现问题，而且可以尽可能多的分享知识，是一种团队成员喜闻乐见的学习/娱乐形式。

但是随着项目的不同，各个团队使用的版本管理工具都不一样。用惯了 `git` 的非常漂亮的 `diff` 子命令之后， `svn` 的 `diff` 简直就是战五渣。没有高亮，没有进度条，就是黑底白字的一些文本，实在无法让人提起兴趣。

这篇文章分享一个简单的方法，可以让你很容易的把 `svn` 的 `diff` 打造成一个漂亮的工具：

```
jtqiu — cat /Users/jtqiu/special-offers.patch | dos2unix | colordiff | less -s -M +Gg — cat — less — 130×41
Index: server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java
===================================================================
--- server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java       (revision 251005)
+++ server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java       (working copy)
@@ -1,5 +1,7 @@
 package com.telstra.tshop.services.client.rightnow;

+import com.google.common.base.Function;
+import com.google.common.base.Joiner;
 import com.telstra.tshop.notification.transformer.BundleBohOrderEmailDOTransformer;
 import com.telstra.tshop.services.shopbox.data.OrderLineItemDO;
 import com.telstra.tshop.services.shopbox.data.online.bundle.PRODUCT_NAMES;
@@ -8,11 +10,15 @@

 import java.util.List;

+import static com.google.common.collect.Lists.newArrayList;
+import static com.google.common.collect.Lists.transform;
+
 public class RightNowLineItemToYesNoConvertor implements RightNowDataConvertor<List<OrderLineItemDO>> {
     public static final String OFFER_CODE_LABEL = "Special Offer Code: ";
     public static final String OFFER_NAME_LABEL = "Special Offer Name: ";
     public static final String OFFER_CATEGORY_LABEL = "Special Offer Category: ";
     public static final String UNAVAILABLE_TEXT = "N/A";
+    private static final char SEPARATOR = '|';


     @Override
@@ -94,36 +100,18 @@

     public String getSpecialOffers(List<OrderLineItemDO> lineItems) {
         if (isEmpty(lineItems)) return null;
-        StringBuilder sb = new StringBuilder();
+
         List<OrderLineItemDO> specialOffers = BundleBohOrderEmailDOTransformer.getPromoSpecialOffers(lineItems);
-        for (OrderLineItemDO item : specialOffers) {
-            if (LineItemTypeDO.PROMOTION.equals(item.getLineItemType())) {
-
-                sb.append(OFFER_CODE_LABEL).append(StringUtils.defaultIfEmpty(item.getOrderLineItemDetails().getProductCode(), UN
AVAILABLE_TEXT));
lines 1-39/193 21%
```

diff格式

Diff是一种通用的表示文本差异的格式，细节可以看我之前写过一篇关于diff和patch的文章。需要说明的是，它作为一种标准格式，很多编辑器都提供对这种格式的高亮显示，比如现在非常流行的 `Sublime Text` 编辑器：

```
Index: server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineIte
===================================================================
--- server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemTo
+++ server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemTo
@@ -1,5 +1,7 @@
 package com.telstra.tshop.services.client.rightnow;

+import com.google.common.base.Function;
+import com.google.common.base.Joiner;
 import com.telstra.tshop.notification.transformer.BundleBohOrderEmailDOTransformer;
 import com.telstra.tshop.services.shopbox.data.OrderLineItemDO;
 import com.telstra.tshop.services.shopbox.data.online.bundle.PRODUCT_NAMES;
@@ -8,11 +10,15 @@

 import java.util.List;

+import static com.google.common.collect.Lists.newArrayList;
+import static com.google.common.collect.Lists.transform;
+
 public class RightNowLineItemToYesNoConvertor implements RightNowDataConvertor<List<O
     public static final String OFFER_CODE_LABEL = "Special Offer Code: ";
     public static final String OFFER_NAME_LABEL = "Special Offer Name: ";
     public static final String OFFER_CATEGORY_LABEL = "Special Offer Category: ";
     public static final String UNAVAILABLE_TEXT = "N/A";
+    private static final char SEPARATOR = '|';


     @Override
@@ -94,36 +100,18 @@

     public String getSpecialOffers(List<OrderLineItemDO> lineItems) {
         if (isEmpty(lineItems)) return null;
-        StringBuilder sb = new StringBuilder();
+
         List<OrderLineItemDO> specialOffers = BundleBohOrderEmailDOTransformer.getPro
-        for (OrderLineItemDO item : specialOffers) {
```

默认的， `svn` 的diff命令会生成这样朴素的输出：

```
Index: server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java
===================================================================
--- server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java      (revision 251005)
+++ server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java      (working copy)
@@ -1,5 +1,7 @@
 package com.telstra.tshop.services.client.rightnow;

+import com.google.common.base.Function;
+import com.google.common.base.Joiner;
 import com.telstra.tshop.notification.transformer.BundleBohOrderEmailDOTransformer;
 import com.telstra.tshop.services.shopbox.data.OrderLineItemDO;
 import com.telstra.tshop.services.shopbox.data.online.bundle.PRODUCT_NAMES;
@@ -8,11 +10,15 @@

 import java.util.List;

+import static com.google.common.collect.Lists.newArrayList;
+import static com.google.common.collect.Lists.transform;
+
 public class RightNowLineItemToYesNoConvertor implements RightNowDataConvertor<List<OrderLineItemDO>> {
     public static final String OFFER_CODE_LABEL = "Special Offer Code: ";
     public static final String OFFER_NAME_LABEL = "Special Offer Name: ";
     public static final String OFFER_CATEGORY_LABEL = "Special Offer Category: ";
     public static final String UNAVAILABLE_TEXT = "N/A";
+    private static final char SEPARATOR = '|';


     @Override
@@ -94,36 +100,18 @@

     public String getSpecialOffers(List<OrderLineItemDO> lineItems) {
         if (isEmpty(lineItems)) return null;
-        StringBuilder sb = new StringBuilder();
+
         List<OrderLineItemDO> specialOffers = BundleBohOrderEmailDOTransformer.getPromoSpecialOffers(lineItems);
-        for (OrderLineItemDO item : specialOffers) {
-            if (LineItemTypeDO.PROMOTION.equals(item.getLineItemType())) {
-
-                sb.append(OFFER_CODE_LABEL).append(StringUtils.defaultIfEmpty(item.getOrderLineItemDetails().getProductCode(), UN
AVAILABLE_TEXT));
:
```

命令行的diff高亮显示

在Mac下，可以通过 `brew` 来安装一个命令行工具，这个工具可以将 `Diff` 格式高亮显示：

```
$ brew install colordiff
```

有了这个工具，就可以将 `svn` 生成的 `Diff` 格式高亮显示出来：

```
$ svn diff | colordiff
```

```
jtqiu — cat /Users/jtqiu/special-offers.patch | colordiff | less — cat — less — 130×41
Index: server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java
===================================================================
--- server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java      (revision 251005)
+++ server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java      (working copy)
@@ -1,5 +1,7 @@
 package com.telstra.tshop.services.client.rightnow;^M
 ^M
+import com.google.common.base.Function;^M
+import com.google.common.base.Joiner;^M
 import com.telstra.tshop.notification.transformer.BundleBohOrderEmailDOTransformer;^M
 import com.telstra.tshop.services.shopbox.data.OrderLineItemDO;^M
 import com.telstra.tshop.services.shopbox.data.online.bundle.PRODUCT_NAMES;^M
@@ -8,11 +10,15 @@
 ^M
 import java.util.List;^M
 ^M
+import static com.google.common.collect.Lists.newArrayList;^M
+import static com.google.common.collect.Lists.transform;^M
+^M
 public class RightNowLineItemToYesNoConvertor implements RightNowDataConvertor<List<OrderLineItemDO>> {^M
     public static final String OFFER_CODE_LABEL = "Special Offer Code: ";^M
     public static final String OFFER_NAME_LABEL = "Special Offer Name: ";^M
     public static final String OFFER_CATEGORY_LABEL = "Special Offer Category: ";^M
     public static final String UNAVAILABLE_TEXT = "N/A";^M
+    private static final char SEPARATOR = '|';^M
 ^M
 ^M
     @Override^M
@@ -94,36 +100,18 @@
 ^M
     public String getSpecialOffers(List<OrderLineItemDO> lineItems) {^M
         if (isEmpty(lineItems)) return null;^M
-        StringBuilder sb = new StringBuilder();^M
+^M
         List<OrderLineItemDO> specialOffers = BundleBohOrderEmailDOTransformer.getPromoSpecialOffers(lineItems);^M
-        for (OrderLineItemDO item : specialOffers) {^M
-            if (LineItemTypeDO.PROMOTION.equals(item.getLineItemType())) {^M
-^M
-                sb.append(OFFER_CODE_LABEL).append(StringUtils.defaultIfEmpty(item.getOrderLineItemDetails().getProductCode(), UN
AVAILABLE_TEXT));^M
:
```

但是你可能已经发现这些神奇的 `^M` ，这是 `Windows` 系统中的换行符在 `Unix` 类系统中的展示，我们需要将 `Diff` 先转换一次：

```
$ svn diff | dos2unix | colordiff
```

```
jtqiu — cat /Users/jtqiu/special-offers.patch | dos2unix | colordiff | less — cat — less — 130×41
Index: server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java
===================================================================
--- server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java      (revision 251005)
+++ server/src/main/java/com/telstra/tshop/services/client/rightnow/RightNowLineItemToYesNoConvertor.java      (working copy)
@@ -1,5 +1,7 @@
 package com.telstra.tshop.services.client.rightnow;

+import com.google.common.base.Function;
+import com.google.common.base.Joiner;
 import com.telstra.tshop.notification.transformer.BundleBohOrderEmailDOTransformer;
 import com.telstra.tshop.services.shopbox.data.OrderLineItemDO;
 import com.telstra.tshop.services.shopbox.data.online.bundle.PRODUCT_NAMES;
@@ -8,11 +10,15 @@

 import java.util.List;

+import static com.google.common.collect.Lists.newArrayList;
+import static com.google.common.collect.Lists.transform;
+
 public class RightNowLineItemToYesNoConvertor implements RightNowDataConvertor<List<OrderLineItemDO>> {
     public static final String OFFER_CODE_LABEL = "Special Offer Code: ";
     public static final String OFFER_NAME_LABEL = "Special Offer Name: ";
     public static final String OFFER_CATEGORY_LABEL = "Special Offer Category: ";
     public static final String UNAVAILABLE_TEXT = "N/A";
+    private static final char SEPARATOR = '|';


     @Override
@@ -94,36 +100,18 @@

     public String getSpecialOffers(List<OrderLineItemDO> lineItems) {
         if (isEmpty(lineItems)) return null;
-        StringBuilder sb = new StringBuilder();
+
         List<OrderLineItemDO> specialOffers = BundleBohOrderEmailDOTransformer.getPromoSpecialOffers(lineItems);
-        for (OrderLineItemDO item : specialOffers) {
-            if (LineItemTypeDO.PROMOTION.equals(item.getLineItemType())) {
-
-                sb.append(OFFER_CODE_LABEL).append(StringUtils.defaultIfEmpty(item.getOrderLineItemDetails().getProductCode(), UN
AVAILABLE_TEXT));
:
```

如果你的系统中没有 `dos2unix` ，可以用 `brew` 来安装：

```
$ brew install dos2unix unix2dos
```

分页器

*nix系统下有两种分页器： `more` 和 `less` ， `less` 比 `more` 的功能更丰富。 `less` 有很多的参数，我们这里选用了3个常见的：

1. `-s` : 压缩连续的空白行为一行
2. `-M` : 给出更多的提示信息，包含行号，百分比等
3. `+Gg` : 先跳至要查看文件的末尾，再跳至文件开头，这样从less就可以得到整个流的长度，从而计算出正确的百分比。当然如果是单独文件时，less是明确知道文件长度的，但是如果是从流中重定向过来的文本，less无法在开始时就得知长度。

下面这条命令可以将当前目录下的所有 `html` 文件分屏显示，并且在每一屏的最后一行显示百分比等信息：

```
$ cat *.html | less -s -M +Gg
```

放在一起

好了，我们将每个部分都已经讲解了一遍了，现在让我们将这些零件串起来，在svn的 `working copy` 中执行这条命令就可以得到非常漂亮的，分页显示的Diff：

```
$ svn diff | dos2unix | colordiff | less -s -M +Gg
```

当然，还可以用一个 `alias` (别名)来节省敲入的字符数：

```
$ alias sd='svn diff | dos2unix | colordiff | less -s -M +Gg'
```

这样你只需要在当前目录输入：

```
$ sd
```

即可启动这个pipeline了。